ECE 220

Lecture x0013 - 04/02

Linked lists - more examples, Trees

Announcements

Announcements

- Reminders

Announcements

- Reminders
  - Quizzes week after next

Announcements

- Reminders
  - Quizzes week after next
  - Remember to reserve CBTF (should open 04/04)

Announcements

- Reminders
  - Quizzes week after next
  - Remember to reserve CBTF (should open 04/04)
  - MP10 takes time - start early!

Announcements

- Reminders
  - Quizzes week after next
  - Remember to reserve CBTF (should open 04/04)
  - MP10 takes time - start early!
- Recap

Announcements

- Reminders
  - Quizzes week after next
  - Remember to reserve CBTF (should open 04/04)
  - MP10 takes time - start early!
- Recap
  - Linked lists

Announcements

- Reminders
  - Quizzes week after next
    - Remember to reserve CBTF (should open 04/04)
    - MP10 takes time - start early!
- Recap
  - Linked lists
    - Maintaining sorted linked lists

Announcements

- Reminders
  - Quizzes week after next
  - Remember to reserve CBTF (should open 04/04)
  - MP10 takes time - start early!
- Recap
  - Linked lists
  - Maintaining sorted linked lists
  - Implementing stack and queue with linked lists

Recap (before exam)

- Given two ***sorted*** linked lists write a function that takes the two head pointers and returns a pointer to a ***merged*** list
- Sort order **must be maintained.** Basic idea …

Recap (before exam)

- Given two ***sorted*** linked lists write a function that takes the two head pointers and returns a pointer to a ***merged*** list
- Sort order **must be maintained.** Basic idea …
    - Traverse both lists until one of them ends, then copy over the remaining list

Recap (before exam)

- Given two ***sorted*** linked lists write a function that takes the two head pointers and returns a pointer to a ***merged*** list
- Sort order **must be maintained.** Basic idea …
  - Traverse both lists until one of them ends, then copy over the remaining list
  - During traversal add new nodes in sorted order

Recap: Merging sorted lists

Recap: Merging sorted lists

Recap: Merging sorted lists

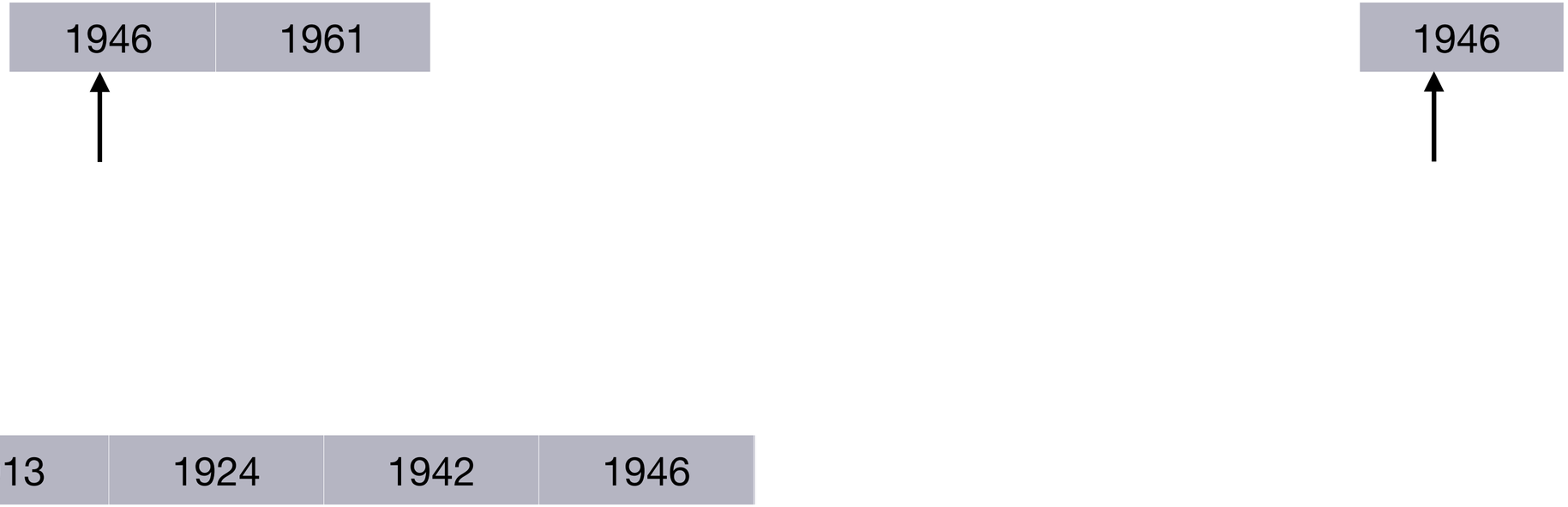
Recap: Merging sorted lists

1911	1913	1924

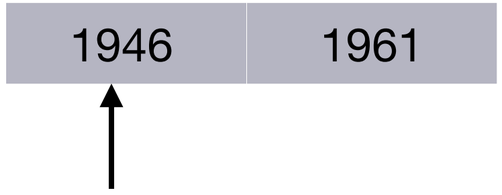
Recap: Merging sorted lists

1911	1913	1924	1942

Recap: Merging sorted lists

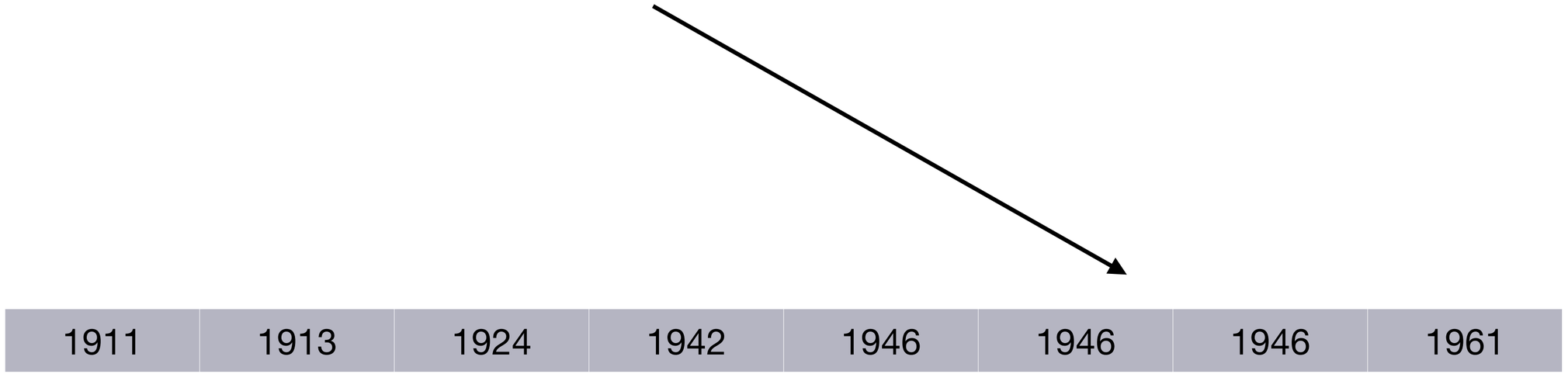


Recap: Merging sorted lists



1911	1913	1924	1942	1946	1946

Recap: Merging sorted lists



Recap: Merging sorted lists

1911	1913	1924	1942	1946	1946	1946	1961

At each step we are adding to tail of the new list …

Recap

Recap

```
node * merge_lists(node *list1, node *list2){
  node *result = NULL;
  if (!list1){
    while (list2){
      add_at_tail(&result, list2);
      list2 = list2->next;
    }
  return result;
}
```

Recap

```
node * merge_lists(node *list1, node *list2){
  node *result = NULL;
  if (!list1){
    while (list2){
      add_at_tail(&result, list2);
      list2 = list2->next;
    }
  return result;
}
```

```
void add_at_tail(node **cursor, node *new){
  if (*cursor == NULL)
    add_at_head(cursor, new);
  else
    add_at_tail(&(*cursor)->next, new);
}
```

Recap

```c
node * merge_lists(node *list1, node *list2){
  node *result = NULL;
  if (!list1){
    while (list2){
      add_at_tail(&result, list2);
      list2 = list2->next;
    }
    return result;
  }
```

```c
void add_at_tail(node **cursor, node *new){
  if (*cursor == NULL)
    add_at_head(cursor, new);
  else
    add_at_tail(&(*cursor)->next, new);
}
```

```c
void add_at_head(node **cursor, node *new){
  node* temp=(node*) malloc(sizeof(node));
  ... /* Copy node to temp */

  if (cursor == NULL)
      *cursor = temp;
  else{
    temp->next = *cursor;
    *cursor = temp;
  }
}
```

Recap

```
node * merge_lists(node *list1, node *list2){
  node *result = NULL;
  if (!list1){
    while (list2){
      add_at_tail(&result, list2);
      list2 = list2->next;
    }
    return result;
  }
  if (!list2){
    while (list1){
      add_at_tail(&result, list1);
      list1 = list1->next;
    }
    return result;
  }
```

```
void add_at_tail(node **cursor, node *new){
  if (*cursor == NULL)
    add_at_head(cursor, new);
  else
    add_at_tail(&(*cursor)->next, new);
}
```

```
void add_at_head(node **cursor, node *new){
  node* temp=(node*) malloc(sizeof(node));
  ... /* Copy node to temp */

  if (cursor == NULL)
      *cursor = temp;
  else{
    temp->next = *cursor;
    *cursor = temp;
  }
}
```

Recap

```c
node * merge_lists(node *list1, node *list2){
  node *result = NULL;
  if (!list1){
    while (list2){
      add_at_tail(&result, list2);
      list2 = list2->next;
    }
    return result;
  }
  if (!list2){
    while (list1){
      add_at_tail(&result, list1);
      list1 = list1->next;
    }
    return result;
  }
```

```c
void add_at_tail(node **cursor, node *new){
  if (*cursor == NULL)
    add_at_head(cursor, new);
  else
    add_at_tail(&(*cursor)->next, new);
}
```

```c
void add_at_head(node **cursor, node *new){
  node* temp=(node*) malloc(sizeof(node));
  ... /* Copy node to temp */

  if (cursor == NULL)
      *cursor = temp;
  else{
    temp->next = *cursor;
    *cursor = temp;
  }
}
```

```c
if (list1->byear <= list2->byear){
  add_at_tail(&result, list1);
  result->next = merge_lists(list1->next, list2);
  }
```

Recap

```c
node * merge_lists(node *list1, node *list2){
  node *result = NULL;
  if (!list1){
    while (list2){
      add_at_tail(&result, list2);
      list2 = list2->next;
    }
    return result;
  }
  if (!list2){
    while (list1){
      add_at_tail(&result, list1);
      list1 = list1->next;
    }
    return result;
  }
```

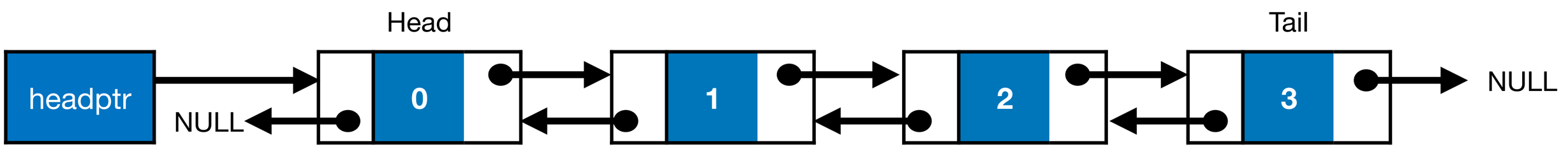
```c
void add_at_tail(node **cursor, node *new){
  if (*cursor == NULL)
    add_at_head(cursor, new);
  else
    add_at_tail(&(*cursor)->next, new);
}
```

```c
void add_at_head(node **cursor, node *new){
  node* temp=(node*) malloc(sizeof(node));
  ... /* Copy node to temp */

  if (cursor == NULL)
      *cursor = temp;
  else{
    temp->next = *cursor;
    *cursor = temp;
  }
}
```

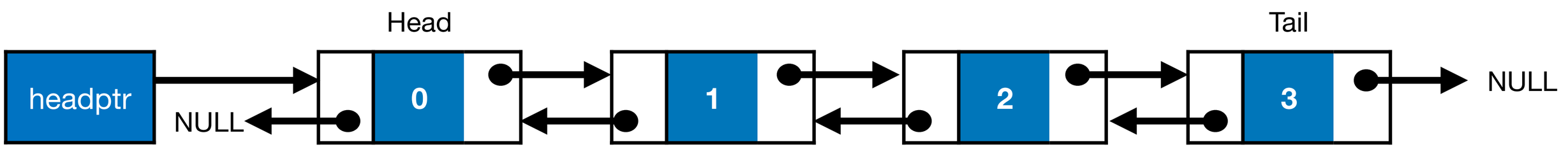
```c
  if (list1->byear <= list2->byear){
    add_at_tail(&result, list1);
    result->next = merge_lists(list1->next, list2);
    }
  else{
    add_at_tail(&result, list2);
    result->next = merge_lists(list1, list2->next);
  }
  return result;
}
```

Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

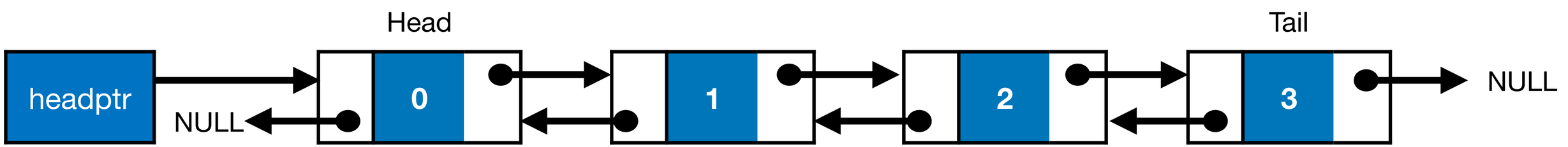
Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

- Advantages:

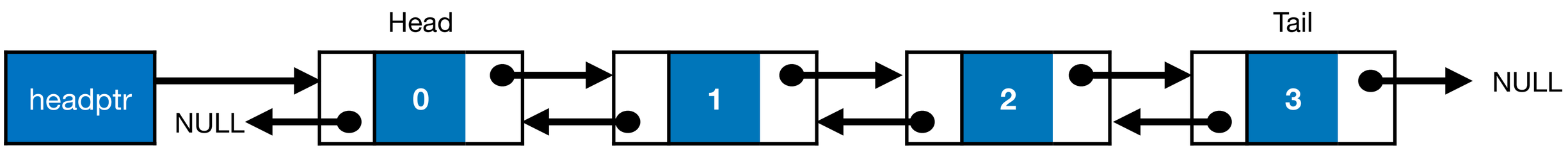
Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

- Advantages:
  - Allows backward and forward traversal

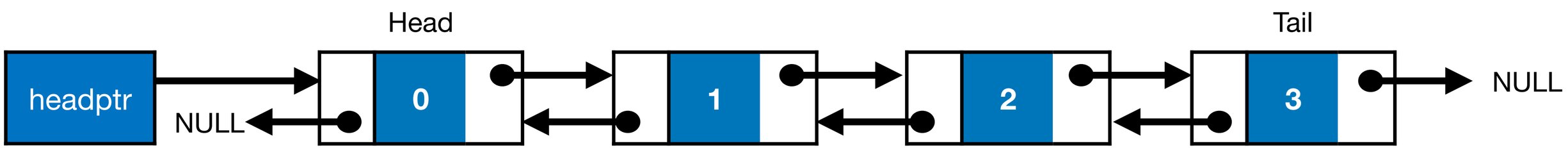
Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

- Advantages:
  - Allows backward and forward traversal
  - Easier to delete a node - why?

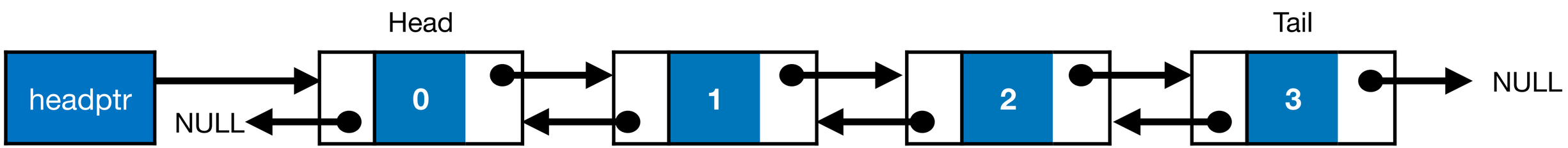
Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

- Advantages:
  - Allows backward and forward traversal
  - Easier to delete a node - why?
- Disadvantages

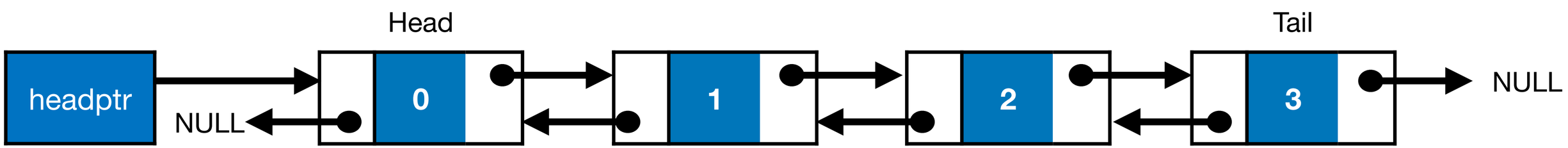
Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

- Advantages:
  - Allows backward and forward traversal
  - Easier to delete a node - why?
- Disadvantages
  - Takes up more memory.

Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

- Advantages:
  - Allows backward and forward traversal
  - Easier to delete a node - why?
- Disadvantages
  - Takes up more memory.
  - Increased bookkeeping, therefore performance overhead

Doubly linked lists

Doubly linked lists

- Updated struct definition

Doubly linked lists

- Updated struct definition

```
typedef struct person{
  char *name;
  unsigned int byear;
  struct person *next;
  struct person *prev;
}node;
```

Doubly linked lists

- Updated struct definition
- Need to modify insertion/deletion functions so that `prev` and `next` are maintained.

```
typedef struct person{
  char *name;
  unsigned int byear;
  struct person *next;
  struct person *prev;
}node;
```

Doubly linked lists

- Updated struct definition
- Need to modify insertion/deletion functions so that `prev` and `next` are maintained.
  - Insertion at head/tail

```
typedef struct person{
  char *name;
  unsigned int byear;
  struct person *next;
  struct person *prev;
}node;
```

Doubly linked lists

- Updated struct definition
- Need to modify insertion/deletion functions so that `prev` and `next` are maintained.
  - Insertion at head/tail
  - Deleting an element

```
typedef struct person{
  char *name;
  unsigned int byear;
  struct person *next;
  struct person *prev;
}node;
```

Doubly linked lists

- Updated struct definition
- Need to modify insertion/deletion functions so that `prev` and `next` are maintained.
  - Insertion at head/tail
  - Deleting an element
  - …

```
typedef struct person{
  char *name;
  unsigned int byear;
  struct person *next;
  struct person *prev;
}node;
```

Insert at head

Insert at head

- Copy `new` into new memory location `temp`; set `temp.prev` to `NULL`

Insert at head

- Copy `new` into new memory location `temp`; set `temp.prev` to `NULL`

```
void add_at_head(node **cursor, node *new){
  node* temp=(node*) malloc(sizeof(node));
  temp->name=new->name;
  temp->byear=new->byear;
  temp->prev = NULL;
```

Insert at head

- Copy `new` into new memory location `temp`; set `temp.prev` to `NULL`
- If list empty then, set `temp.next` to `NULL` and head to `temp`

```
void add_at_head(node **cursor, node *new){
  node* temp=(node*) malloc(sizeof(node));
  temp->name=new->name;
  temp->byear=new->byear;
  temp->prev = NULL;

if (*cursor == NULL){
  temp->next = NULL;
  *cursor = temp;
}
```

Insert at head

- Copy `new` into new memory location `temp`; set `temp.prev` to `NULL`
- If list empty then, set `temp.next` to `NULL` and head to `temp`
- Else do bookkeeping

```
void add_at_head(node **cursor, node *new){
  node* temp=(node*) malloc(sizeof(node));
  temp->name=new->name;
  temp->byear=new->byear;
  temp->prev = NULL;

if (*cursor == NULL){
  temp->next = NULL;
  *cursor = temp;
}
else{
```

Insert at head

- Copy `new` into new memory location `temp`; set `temp.prev` to `NULL`
- If list empty then, set `temp.next` to `NULL` and head to `temp`
- Else do bookkeeping
  - `temp.next` is current head

```
void add_at_head(node **cursor, node *new){
  node* temp=(node*) malloc(sizeof(node));
  temp->name=new->name;
  temp->byear=new->byear;
  temp->prev = NULL;

if (*cursor == NULL){
  temp->next = NULL;
  *cursor = temp;
}
else{
  temp->next=*cursor;
```

Insert at head

- Copy `new` into new memory location `temp`; set `temp.prev` to `NULL`
- If list empty then, set `temp.next` to `NULL` and head to `temp`
- Else do bookkeeping
  - `temp.next` is current head
  - current head's `prev` is `temp`

```
void add_at_head(node **cursor, node *new){
  node* temp=(node*) malloc(sizeof(node));
  temp->name=new->name;
  temp->byear=new->byear;
  temp->prev = NULL;

if (*cursor == NULL){
  temp->next = NULL;
  *cursor = temp;
}
else{
  temp->next=*cursor;
  (*cursor)->prev = temp;
```

Insert at head

- Copy `new` into new memory location `temp`; set `temp.prev` to `NULL`
- If list empty then, set `temp.next` to `NULL` and head to `temp`
- Else do bookkeeping
  - `temp.next` is current head
  - current head's `prev` is `temp`
  - New head is `temp`

```
void add_at_head(node **cursor, node *new){
  node* temp=(node*) malloc(sizeof(node));
  temp->name=new->name;
  temp->byear=new->byear;
  temp->prev = NULL;
  if (*cursor == NULL){
    temp->next = NULL;
    *cursor = temp;
  }
  else{
    temp->next=*cursor;
    (*cursor)->prev = temp;
    *cursor = temp;
  }
}
```

Insert at tail

```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)

  if ((*cursor)->next==NULL){
  }
  else

}
```

Insert at tail

- If empty list, call `add_at_head` on cursor

```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
    }
    else

}
```

Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.

```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
    add_at_head(&(*cursor)->next, new);
    }
  else

}
```

Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element

```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
    add_at_head(&(*cursor)->next, new);
    (*cursor)->next->prev = *cursor;
  }
  else

}
```

Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element
- Otherwise …
- recurse until at at last element

```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
    add_at_head(&(*cursor)->next, new);
    (*cursor)->next->prev = *cursor;
  }
  else
    add_at_tail(&(*cursor)->next, new);
}
```

Insert at tail

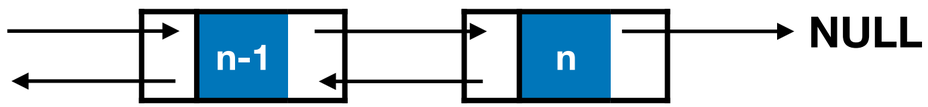
- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element
- Otherwise …
- recurse until at at last element

```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
  }
  else
    add_at_tail(&(*cursor)->next, new);
}
```

Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element
- Otherwise …
- recurse until at at last element

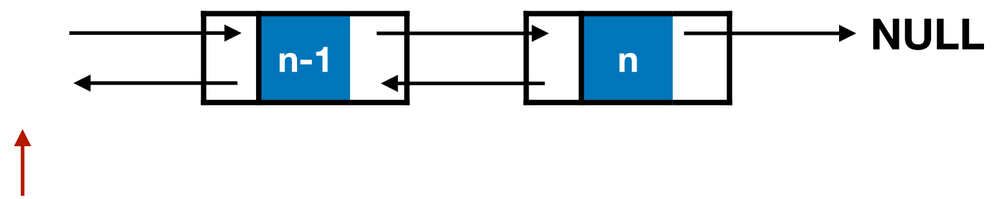
```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
  }
  else
    add_at_tail(&(*cursor)->next, new);
}
```



Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element
- Otherwise …
- recurse until at at last element

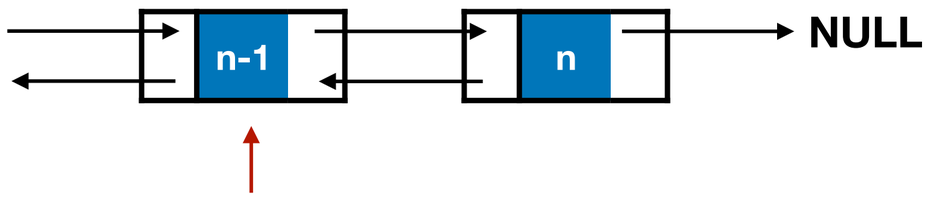
```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
  }
  else
    add_at_tail(&(*cursor)->next, new);
}
```



Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element
- Otherwise …
- recurse until at at last element

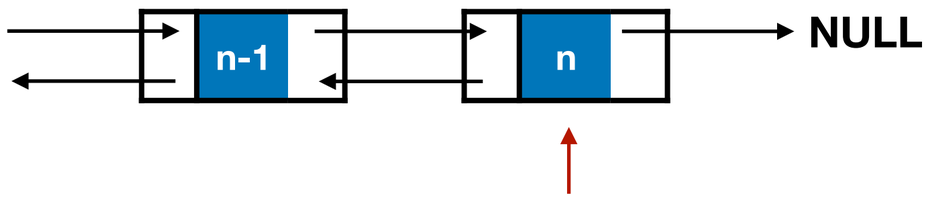
```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
  }
  else
    add_at_tail(&(*cursor)->next, new);
}
```



Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element
- Otherwise …
- recurse until at at last element

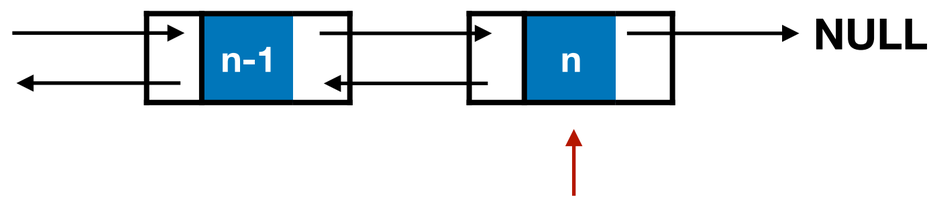
```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
  }
  else
    add_at_tail(&(*cursor)->next, new);
}
```



Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element
- Otherwise …
- recurse until at at last element

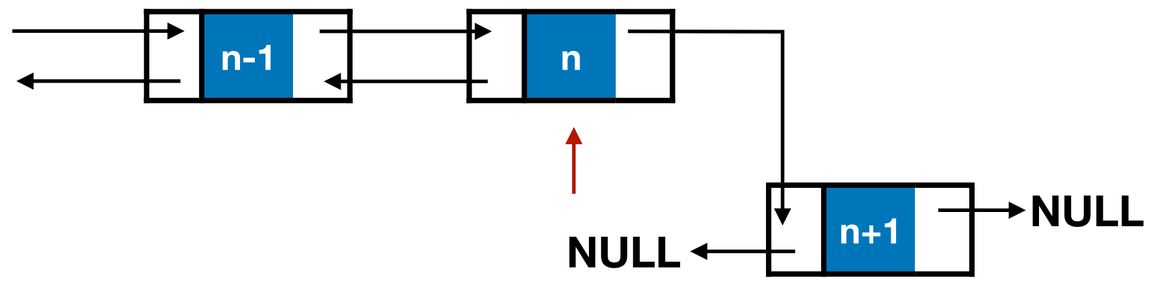
```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
    add_at_head(&(*cursor)->next, new);
  }
  else
    add_at_tail(&(*cursor)->next, new);
}
```



Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element
- Otherwise …
- recurse until at at last element

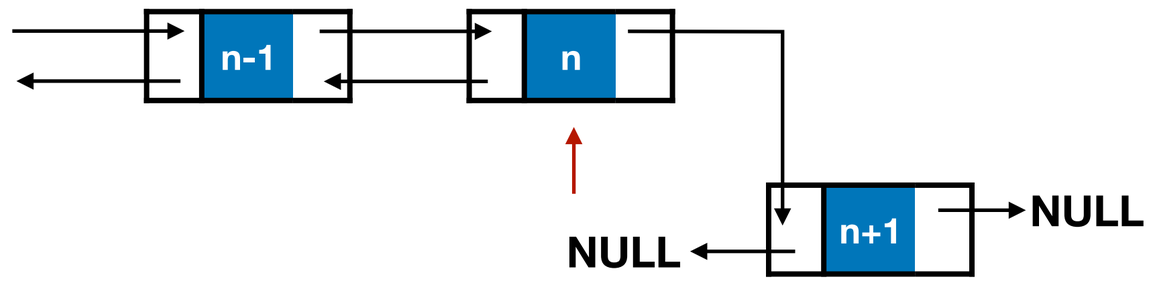
```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
    add_at_head(&(*cursor)->next, new);
  }
  else
    add_at_tail(&(*cursor)->next, new);
}
```



Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element
- Otherwise …
- recurse until at at last element

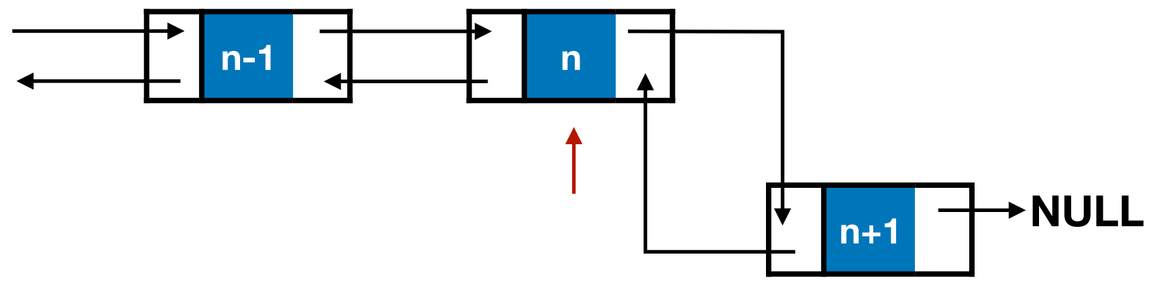
```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
    add_at_head(&(*cursor)->next, new);
    (*cursor)->next->prev = *cursor;
  }
  else
    add_at_tail(&(*cursor)->next, new);
}
```



Insert at tail

- If empty list, call `add_at_head` on cursor
- If at last element, call `add_at_head` on its `next`.
- Then set the new node's prev to last element
- Otherwise …
- recurse until at at last element

```
void add_at_tail(node **cursor, node *new){
  if (*(cursor)==NULL)
    add_at_head(&(*cursor), new);
  if ((*cursor)->next==NULL){
    add_at_head(&(*cursor)->next, new);
    (*cursor)->next->prev = *cursor;
  }
  else
    add_at_tail(&(*cursor)->next, new);
}
```



Delete from head

Delete from head

```
void del_head(node **headptr){
```

Delete from head

- If empty list, do nothing

```
void del_head(node **headptr){
  if (*headptr==NULL)
    return;
```

Delete from head

- If empty list, do nothing
- Otherwise, copy old head pointer

```
void del_head(node **headptr){
  if (*headptr==NULL)
    return;
  else{
    node * old_head = *headptr;
```

Delete from head

- If empty list, do nothing
- Otherwise, copy old head pointer
- Advance head pointer

```
void del_head(node **headptr){
  if (*headptr==NULL)
    return;
  else{
node * old_head = *headptr;
*headptr = (*headptr)->next;
```

Delete from head

- If empty list, do nothing
- Otherwise, copy old head pointer
- Advance head pointer
- Set new head pointer's `prev` to `NULL`

```
void del_head(node **headptr){
  if (*headptr==NULL)
    return;
  else{
node * old_head = *headptr;
*headptr = (*headptr)->next;
(*headptr)->prev = NULL;
```

Delete from head

- If empty list, do nothing
- Otherwise, copy old head pointer
- Advance head pointer
- Set new head pointer's `prev` to `NULL`
- `free` old head pointer

```
void del_head(node **headptr){
  if (*headptr==NULL)
    return;
  else{
    node * old_head = *headptr;
    *headptr = (*headptr)->next;
    (*headptr)->prev = NULL;
    free(old_head);
  }
}
```

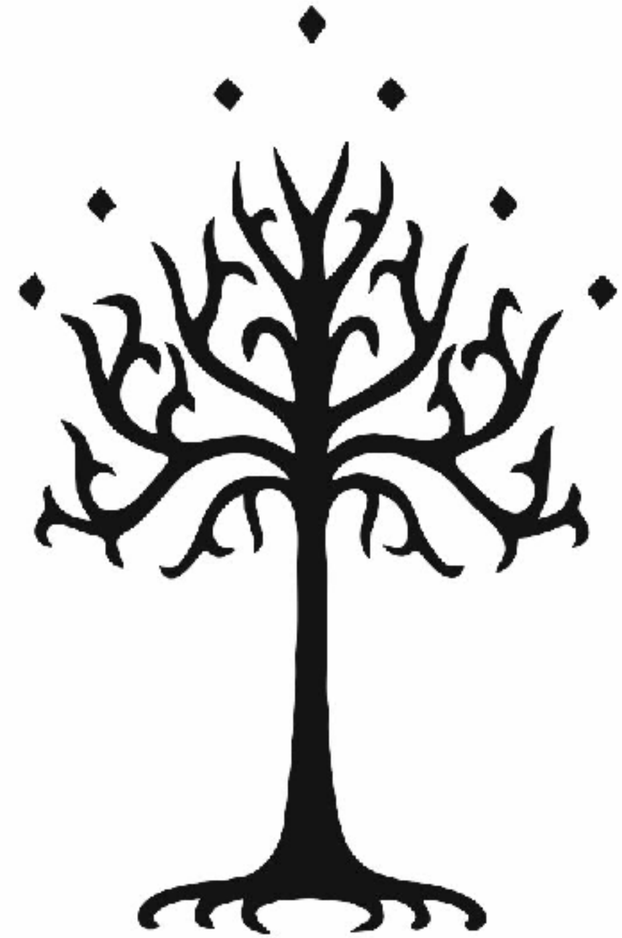
Left as exercise …

- ***Highly encouraged*** to implement for doubly linked lists:
    - Adding a node in the middle
        - Options: (a) maintain sorted list, (b) add before/after given node pointer
    - Deleting: (a) given node (b) tail element

Today

Today

- New data structure: ***trees***



Today

- New data structure: ***trees***

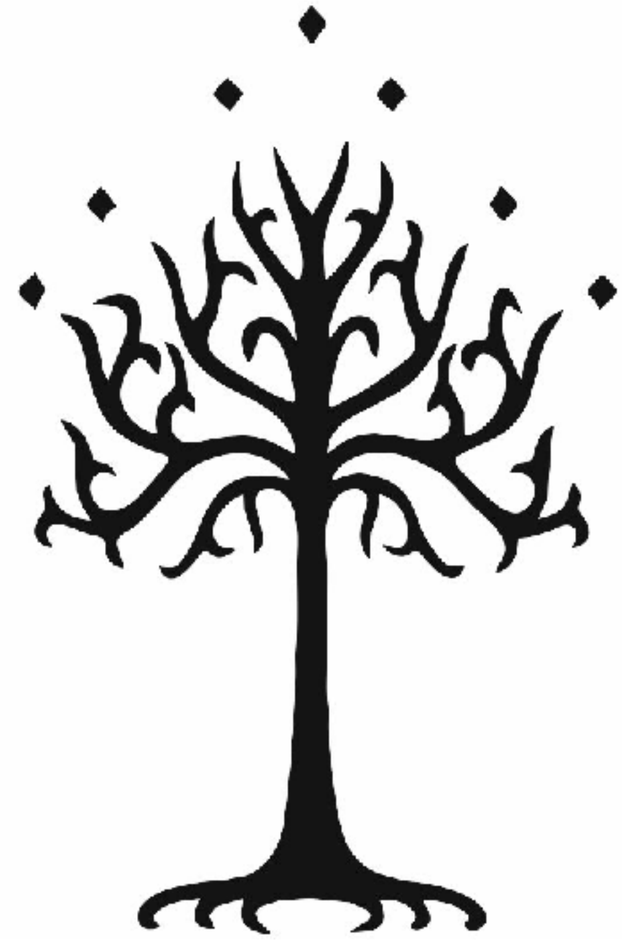
- Linked lists, queues, stacks: *linear* data structures

Today

- New data structure: ***trees***

- Linked lists, queues, stacks: *linear* data structures

- Trees are *nonlinear* & hierarchical



Today

- New data structure: ***trees***

- Linked lists, queues, stacks: *linear* data structures

- Trees are *nonlinear* & hierarchical

- Think familty trees or organizational charts

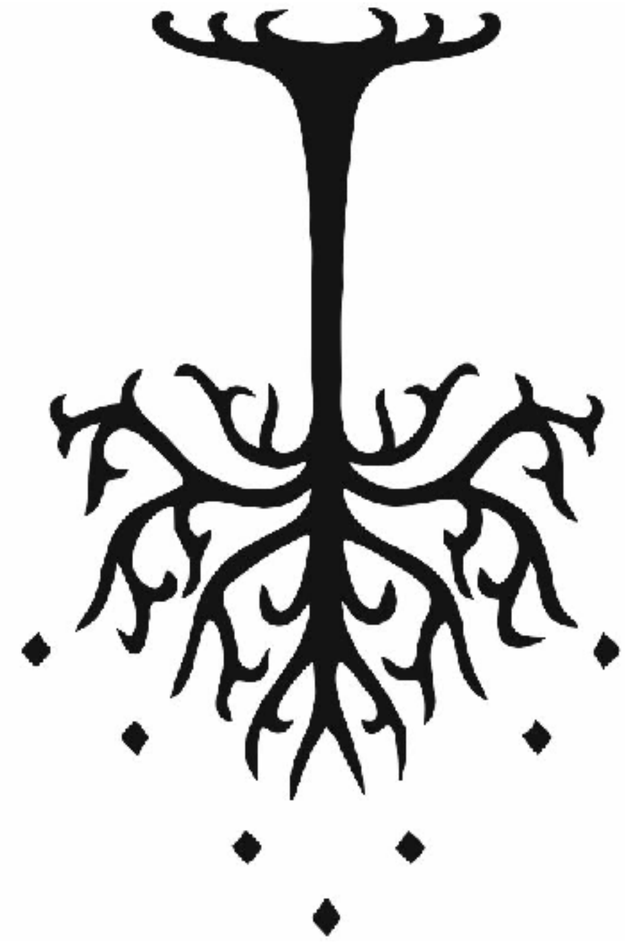
Today

- New data structure: ***trees***

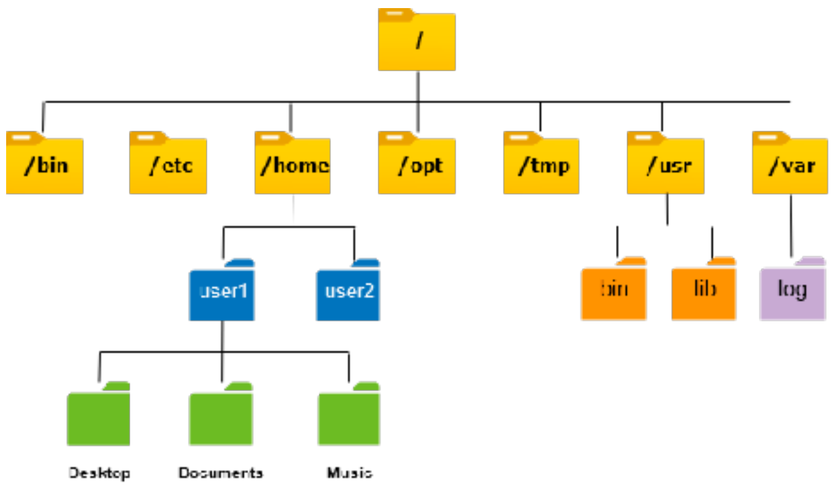
- Linked lists, queues, stacks: *linear* data structures

- Trees are *nonlinear* & hierarchical

- Think familty trees or organizational charts

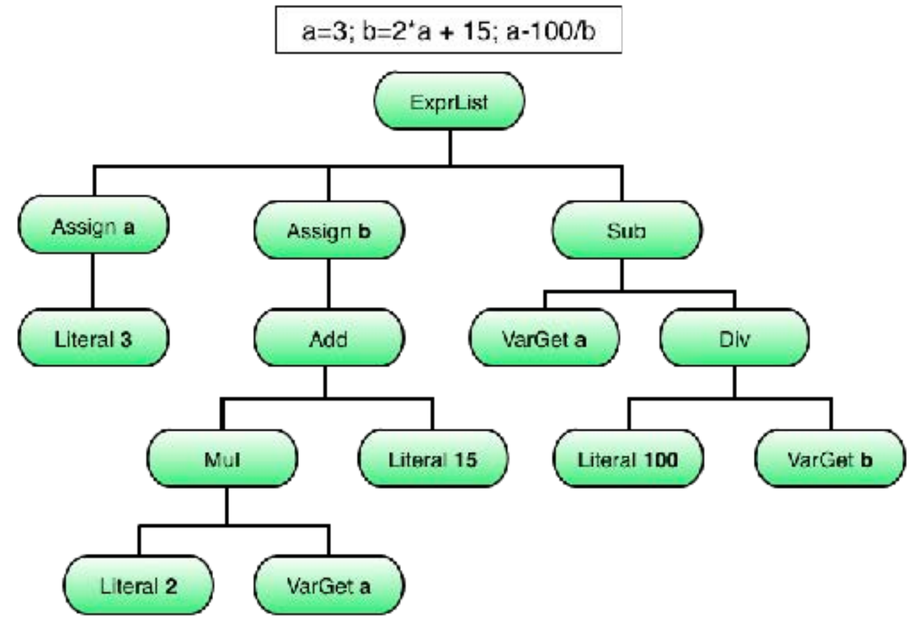


Why trees?

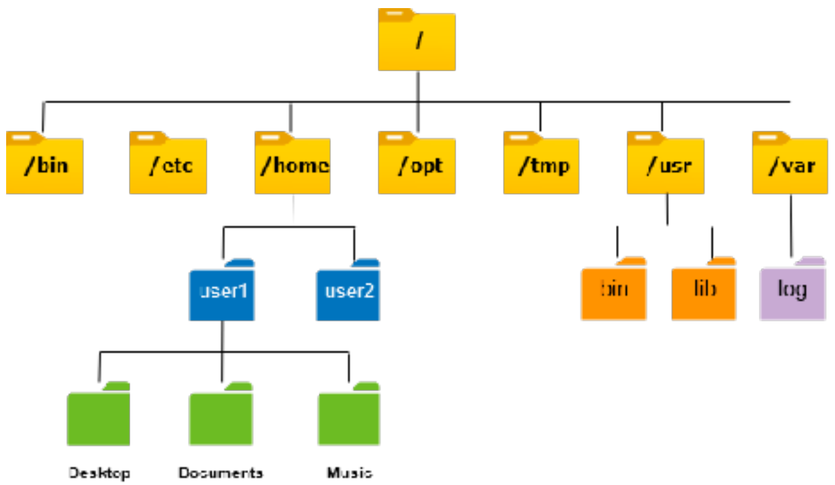


QuadTree: https://en.wikipedia.org/wiki/Quadtree

Filesystems, computer graphics, programming languages, taxonomic classifictaion, etc.

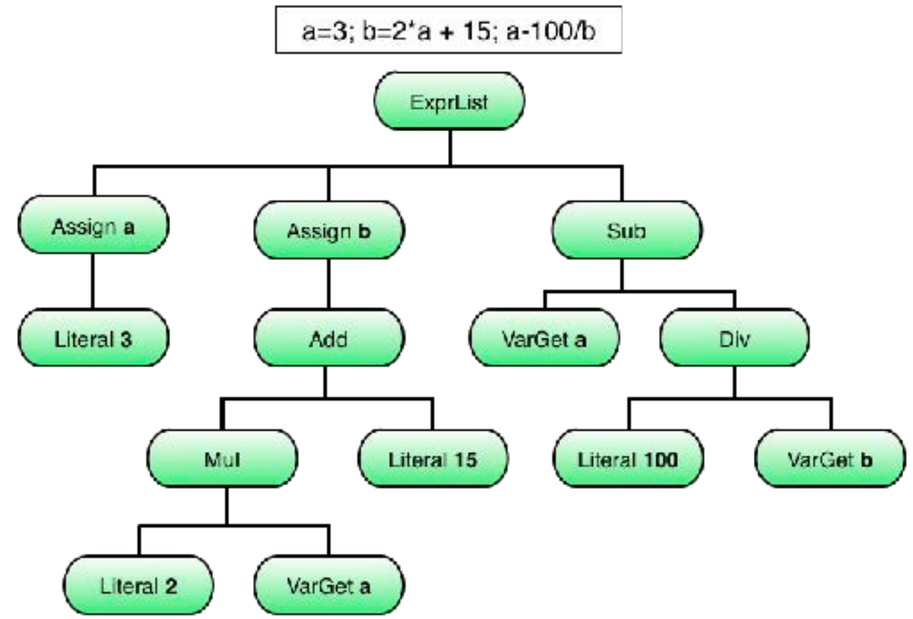


Why trees?

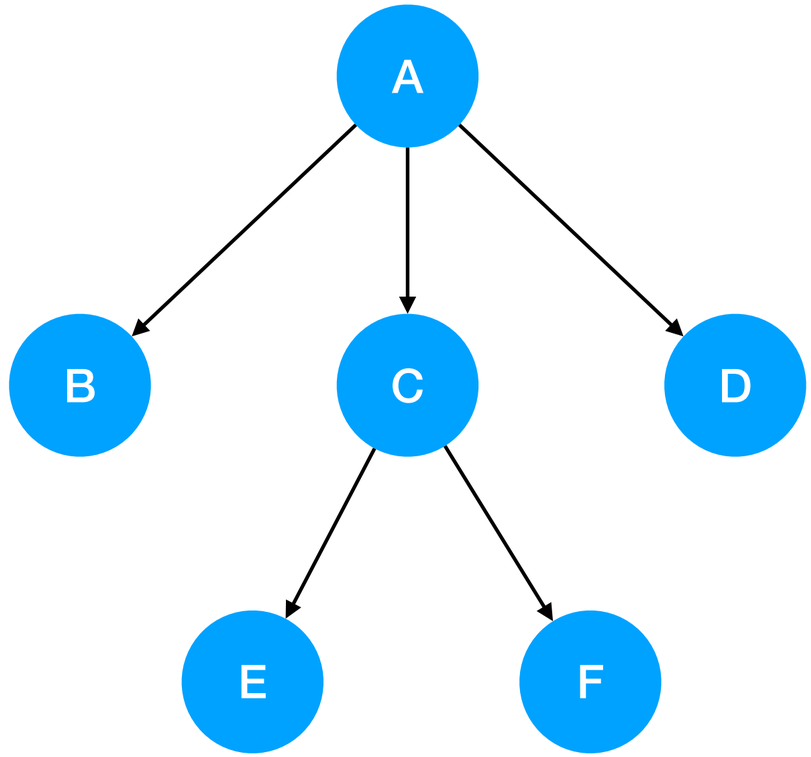


QuadTree: https://en.wikipedia.org/wiki/Quadtree

Filesystems, computer graphics, programming languages, taxonomic classifictaion, etc.

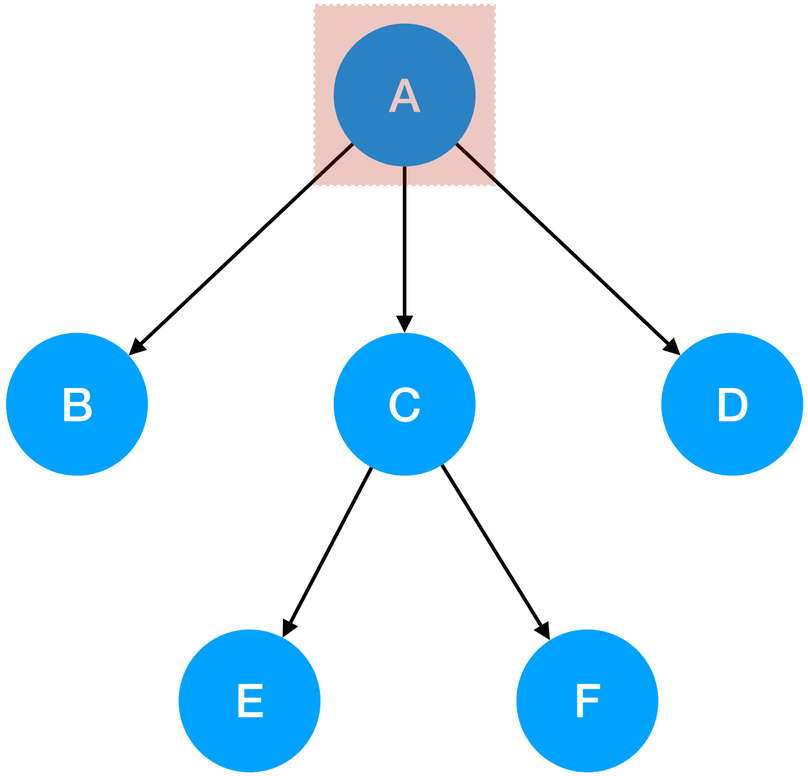


Concepts related to trees



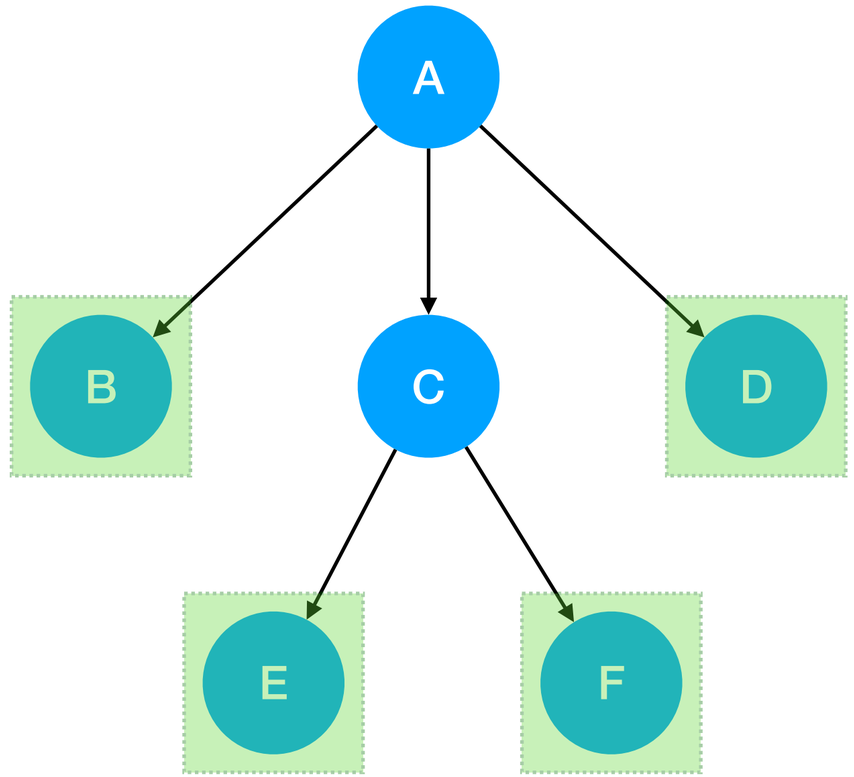
Concepts related to trees

- Root
  - Top most node, no parent.



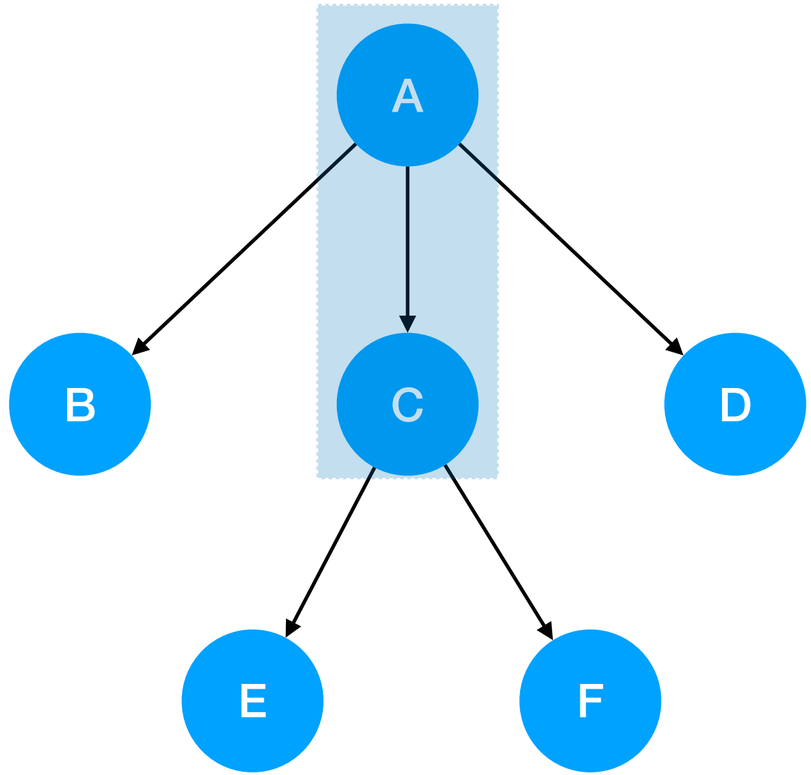
Concepts related to trees

- Root
  - Top most node, no parent.
- Leaf
  - Outermost nodes, no children

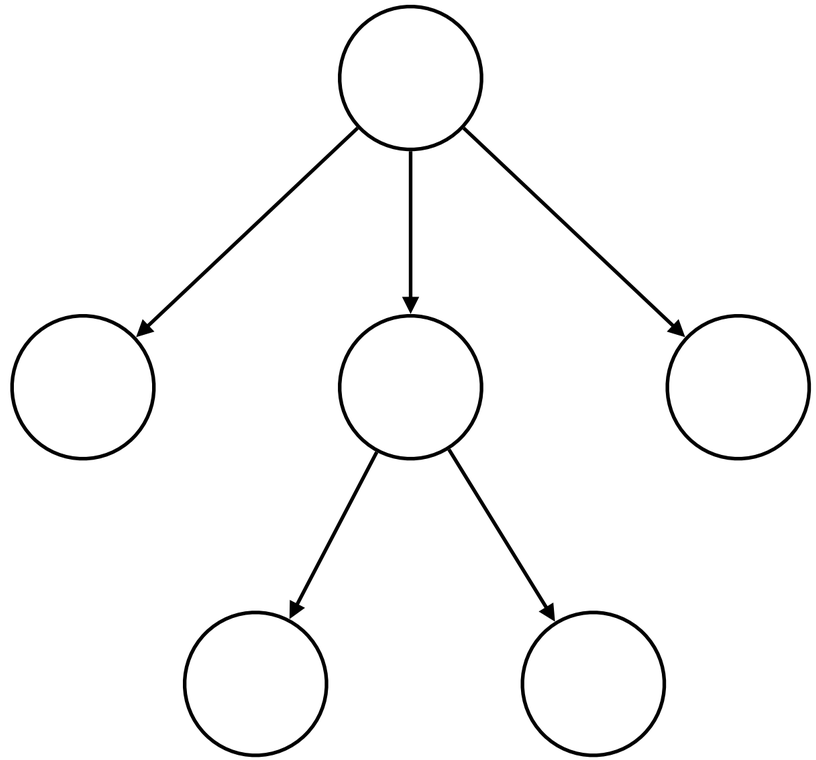


Concepts related to trees

- Root
  - Top most node, no parent.
- Leaf
  - Outermost nodes, no children
- Inner node(s)
  - Has atleast one child

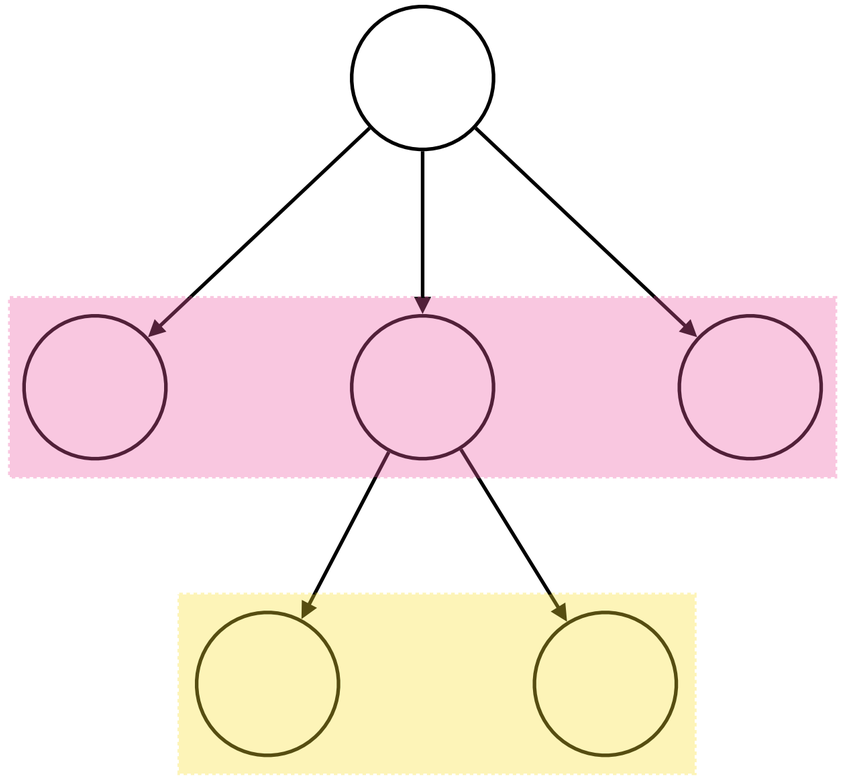


Concepts related to trees



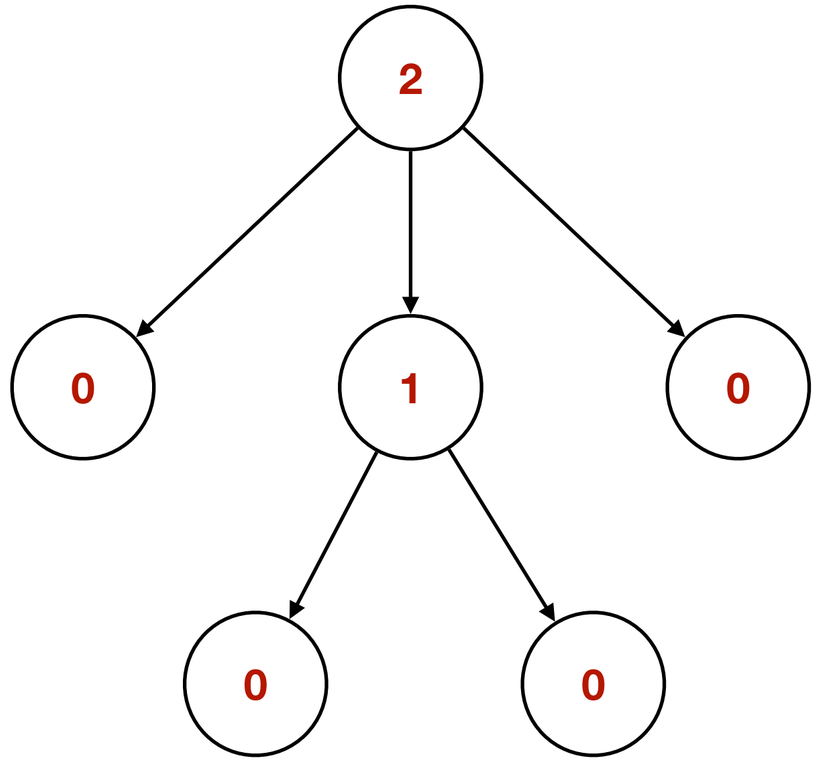
Concepts related to trees

- Siblings



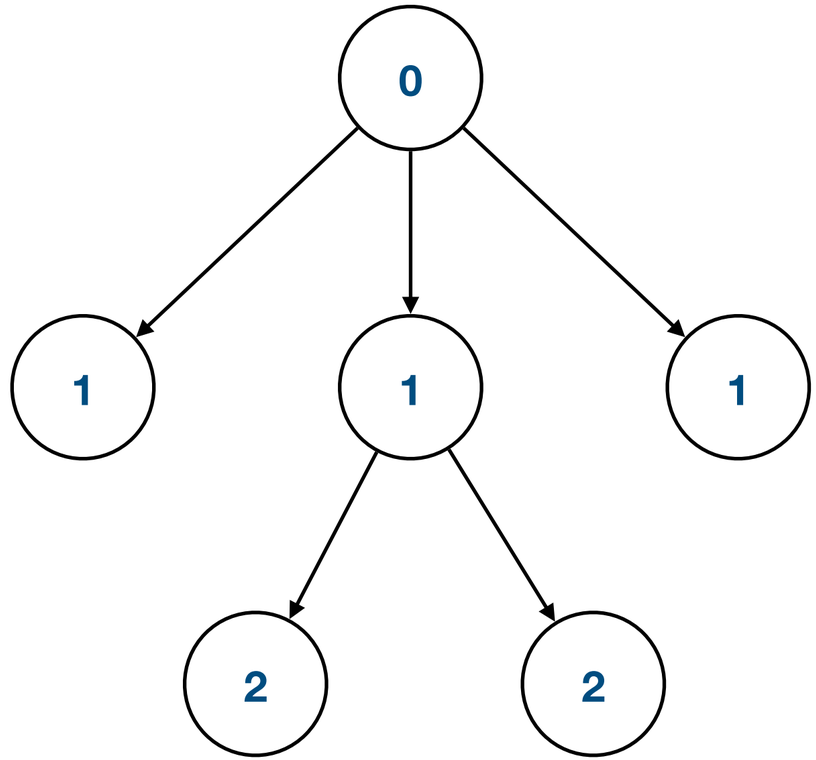
Concepts related to trees

- Siblings
- Height (of a node)
  - Length of *longest* path from given node to a leaf



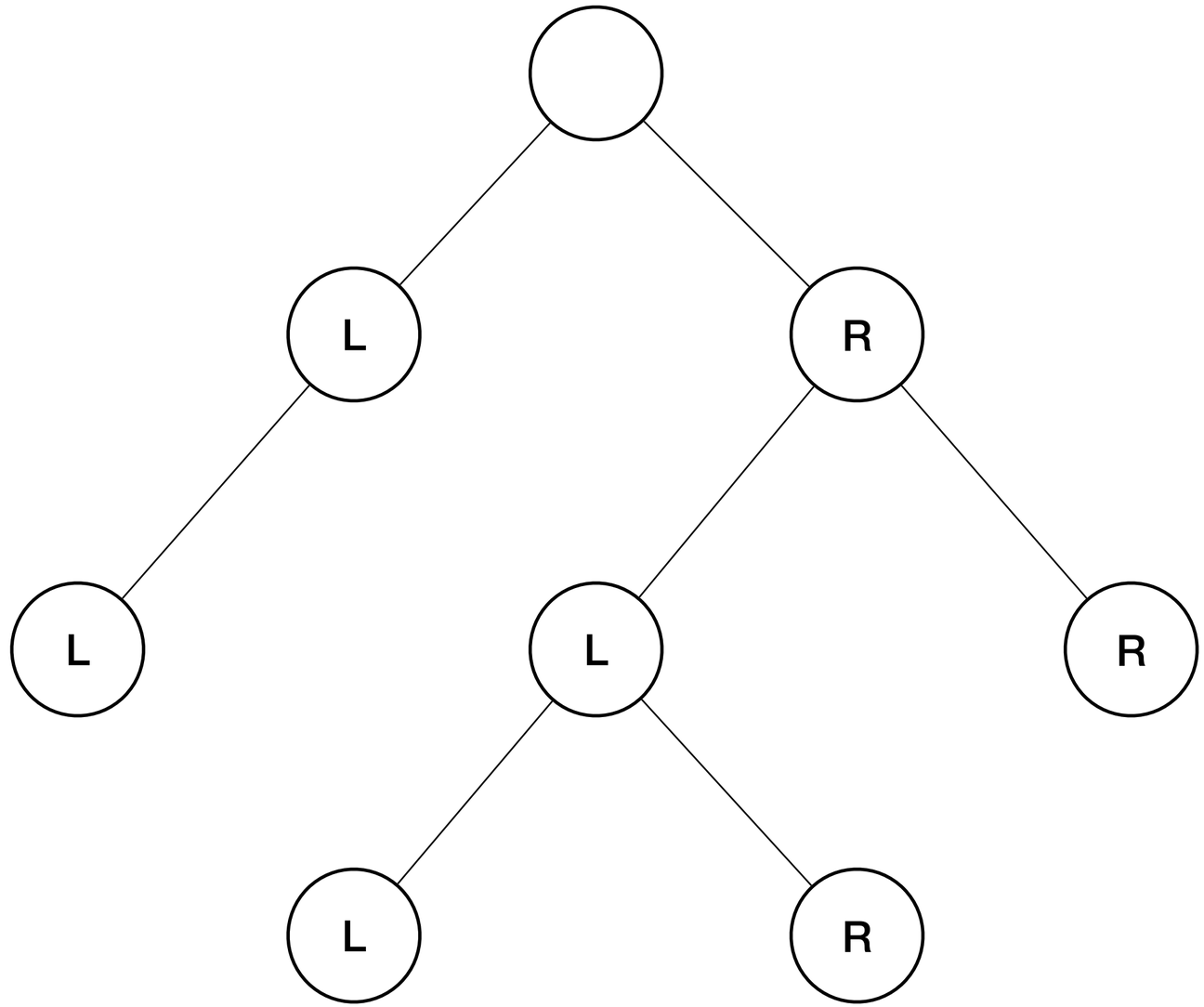
Concepts related to trees

- Siblings
- Height (of a node)
  - Length of *longest* path from given node to a leaf
- Depth (of a node)
  - Length of path from root to given node



Binary trees

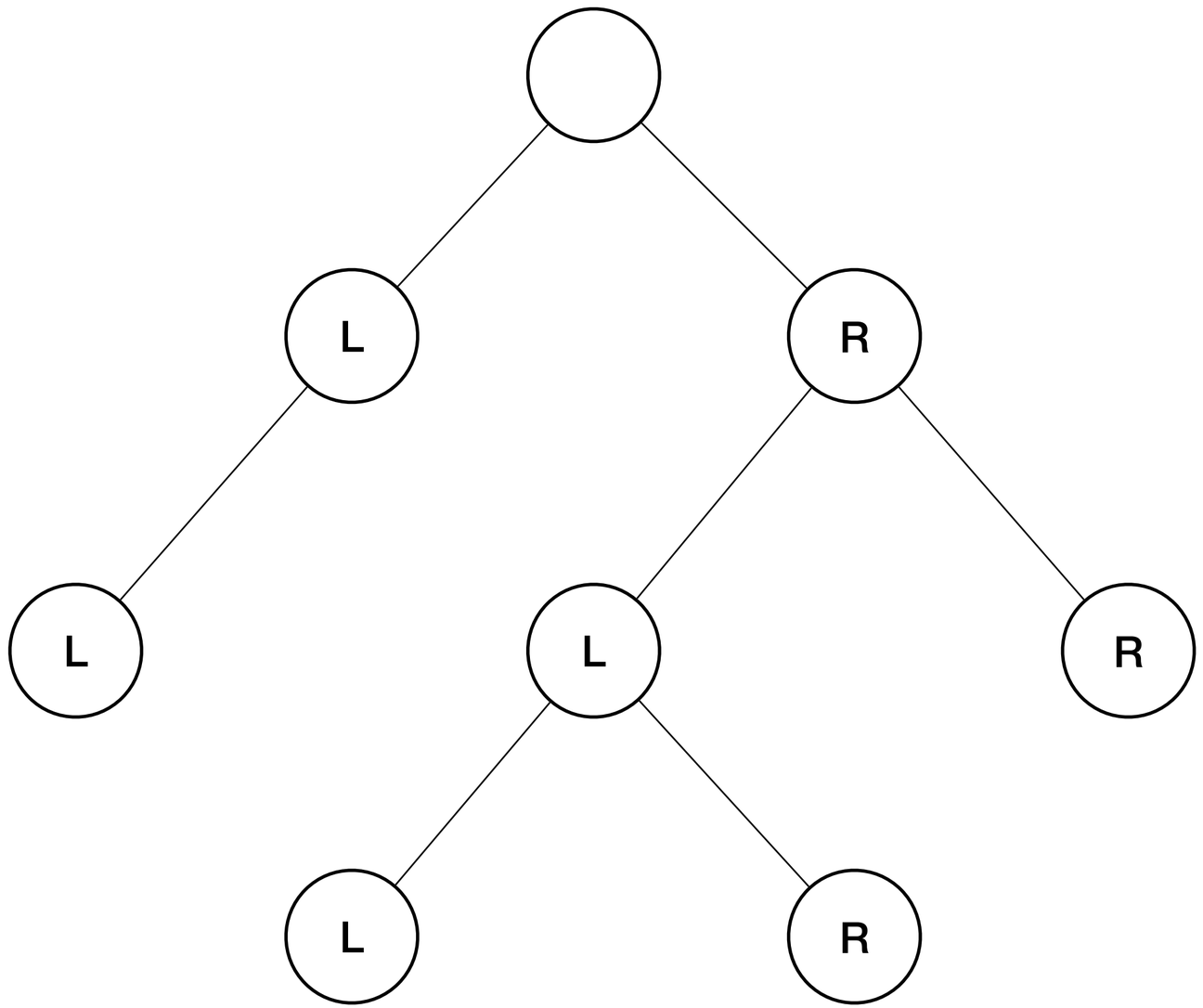
- Trees where every node has *at most* two children.



Binary trees

- Trees where every node has *at most* two children.

```
typedef struct person node;
struct person{
  char *name;
  node *next;
  node *prev;
}node;
```

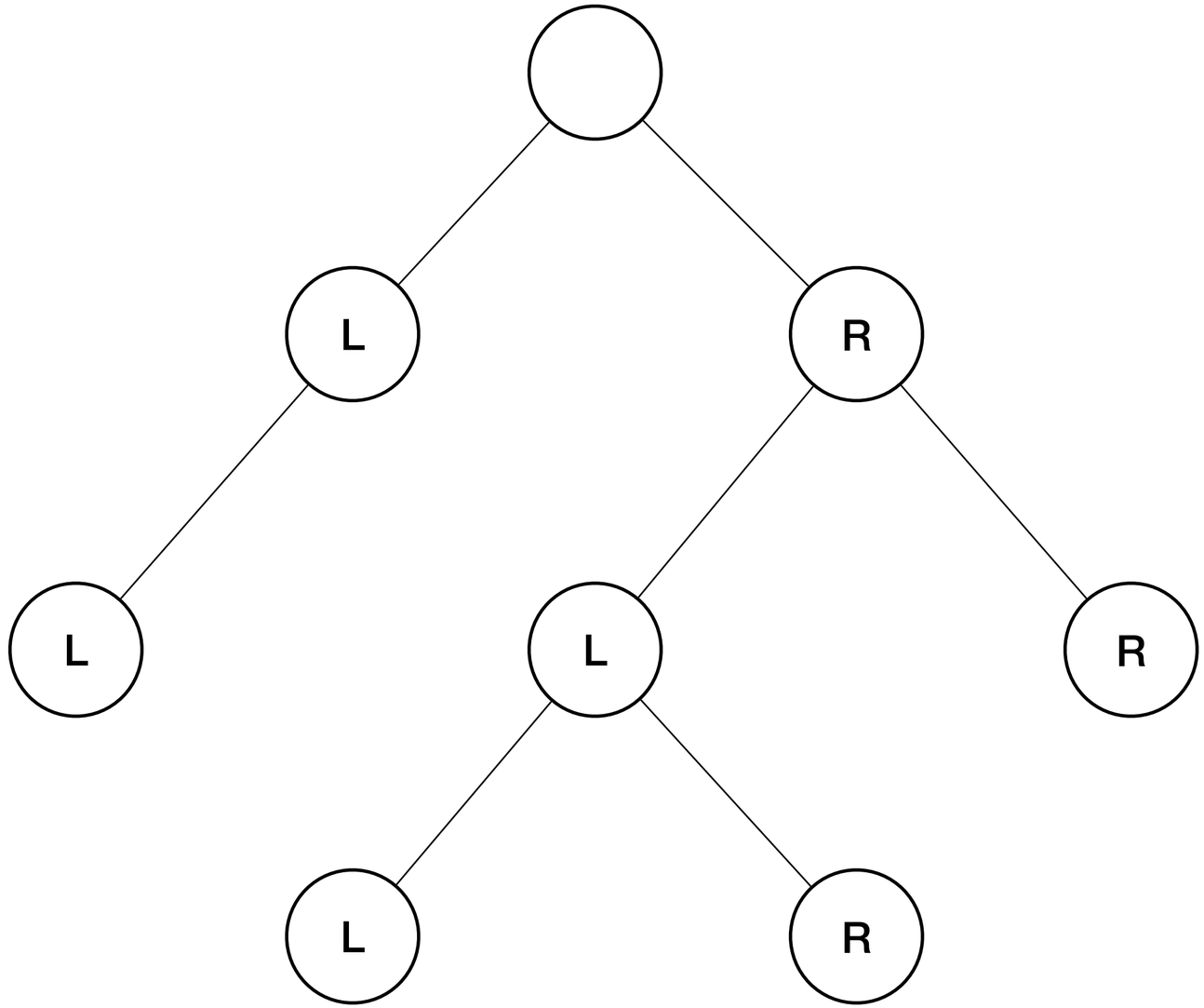


Binary trees

- Trees where every node has *at most* two children.

```
typedef struct person node;
struct person{
   char *name;
   node *next;
   node *prev;
}node;


typedef struct node treeNode;
struct node{
    int data;
    treeNode *left;
    treeNode *right;
};
```



Traversing trees

Traversing trees

- You can traverse trees in three ways

Traversing trees

- You can traverse trees in three ways
  - Pre-order
    - Root, Left, Right

For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

Traversing trees

- You can traverse trees in three ways
  - Pre-order
    - Root, Left, Right
  - In-order
    - Left, Root, Right

For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

For each node, visit the **left** subtree, then read the data of the **node**, then visit the **right** subtree.

Traversing trees

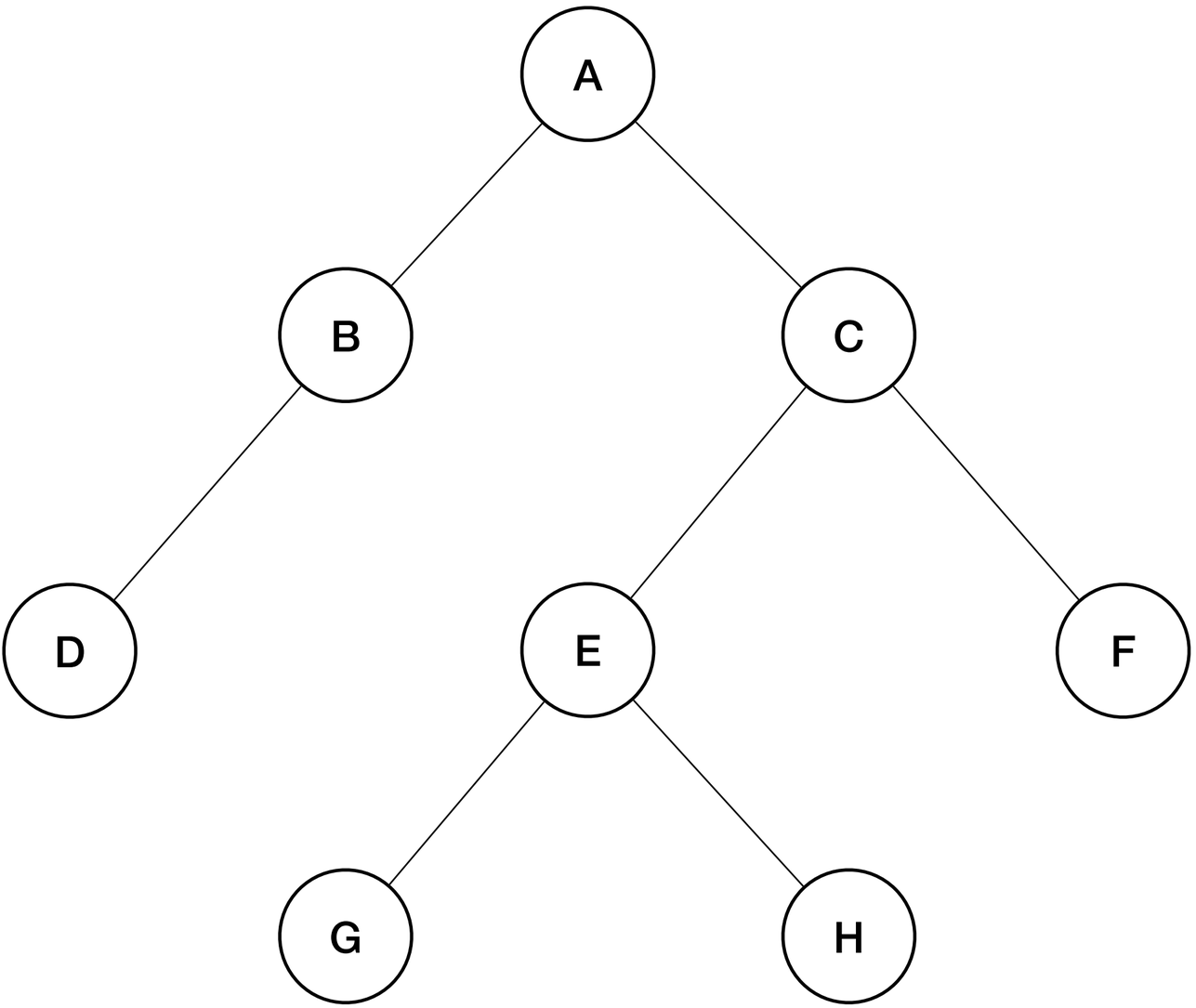
- You can traverse trees in three ways
  - Pre-order
    - Root, Left, Right
  - In-order
    - Left, Root, Right
  - Post-order
    - Left, Right, Root

For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

For each node, visit the **left** subtree, then read the data of the **node**, then visit the **right** subtree.

For each node, visit the **left** subtree, then visit the **right** subtree, then read the data of the **node**.

Traversing trees

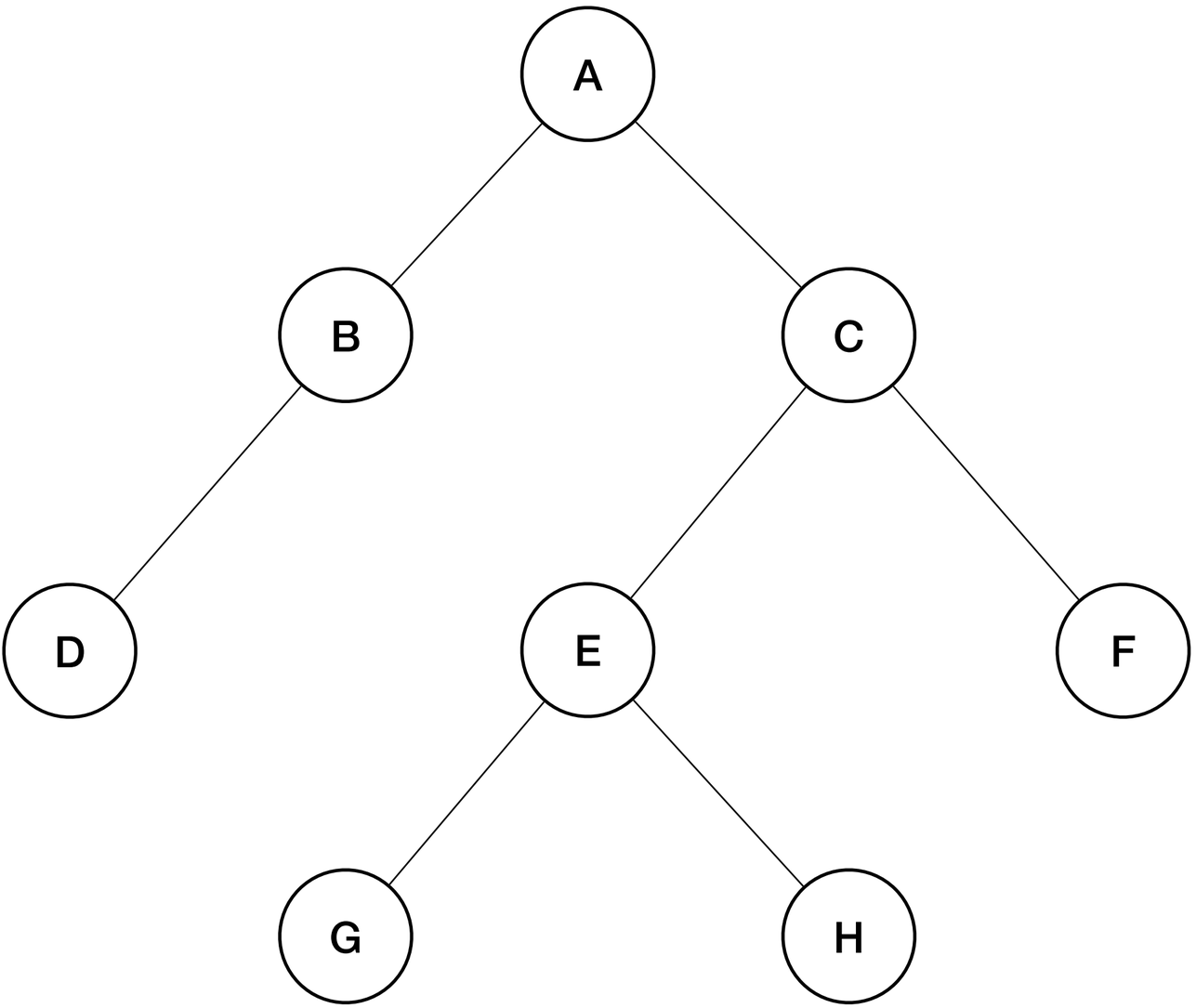


For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

For each node, visit the **left** subtree, then read the data of the **node**, then visit the **right** subtree.

For each node, visit the **left** subtree, then visit the **right** subtree, then read the data of the **node**.

Traversing trees



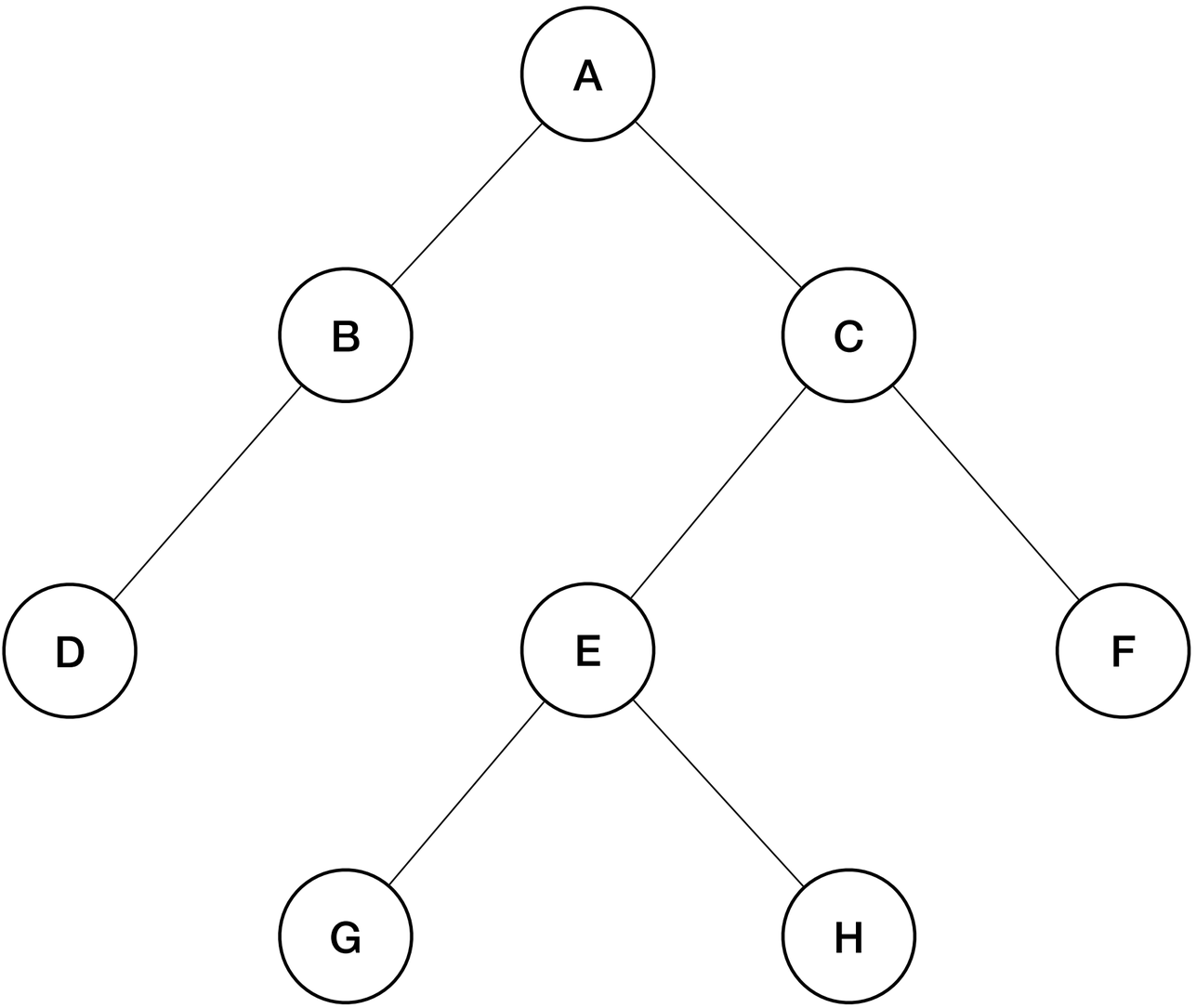
For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

A → B → D → C → E → G → H → F

For each node, visit the **left** subtree, then read the data of the **node**, then visit the **right** subtree.

For each node, visit the **left** subtree, then visit the **right** subtree, then read the data of the **node**.

Traversing trees



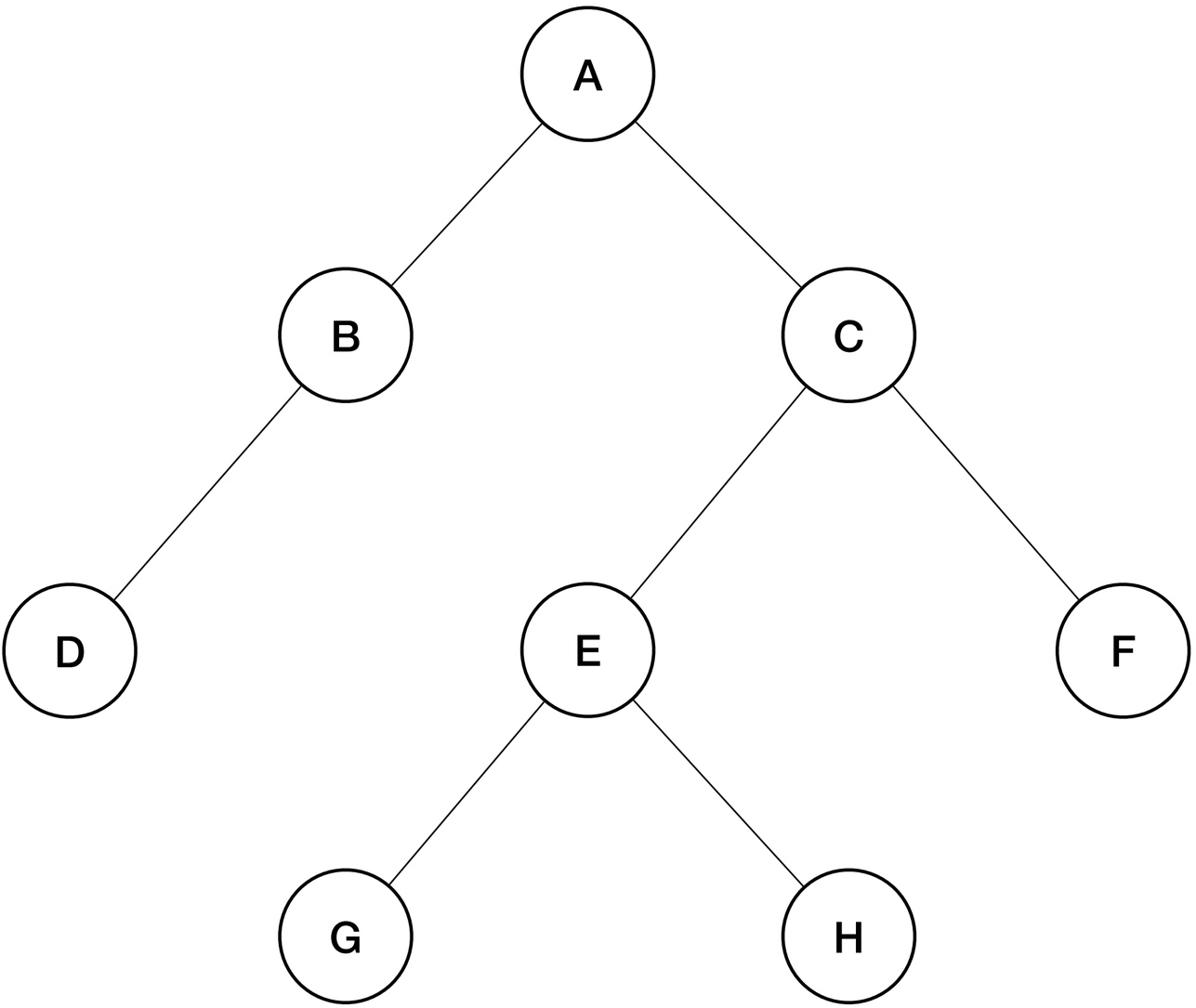
For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

A → B → D → C → E → G → H → F

For each node, visit the **left** subtree, then read the data of the **node**, then visit the **right** subtree.

For each node, visit the **left** subtree, then visit the **right** subtree, then read the data of the **node**.

Traversing trees



For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

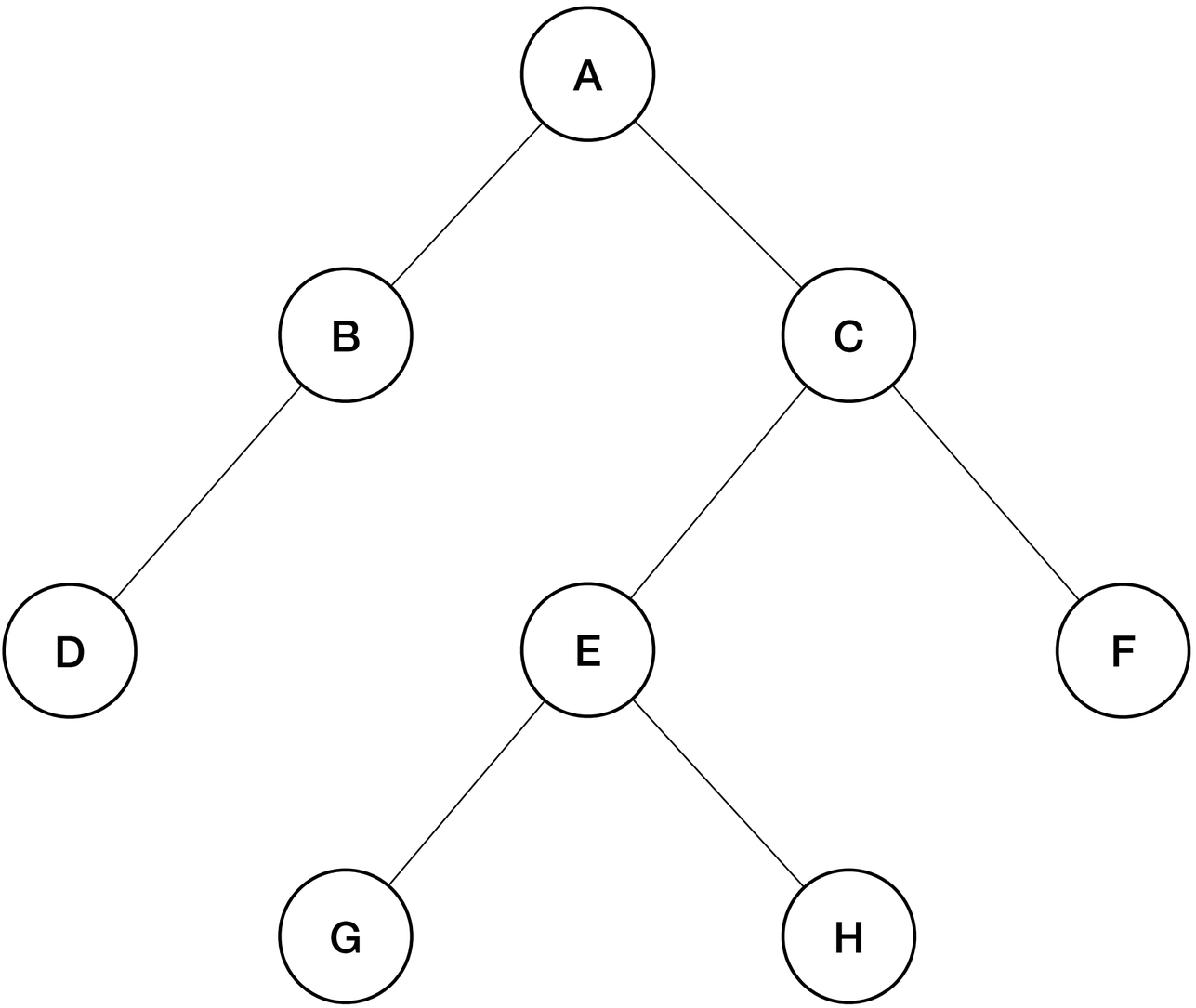
A → B → D → C → E → G → H → F

For each node, visit the **left** subtree, then read the data of the **node**, then visit the **right** subtree.

D → B → A → G → E → H → C → F

For each node, visit the **left** subtree, then visit the **right** subtree, then read the data of the **node**.

Traversing trees



For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

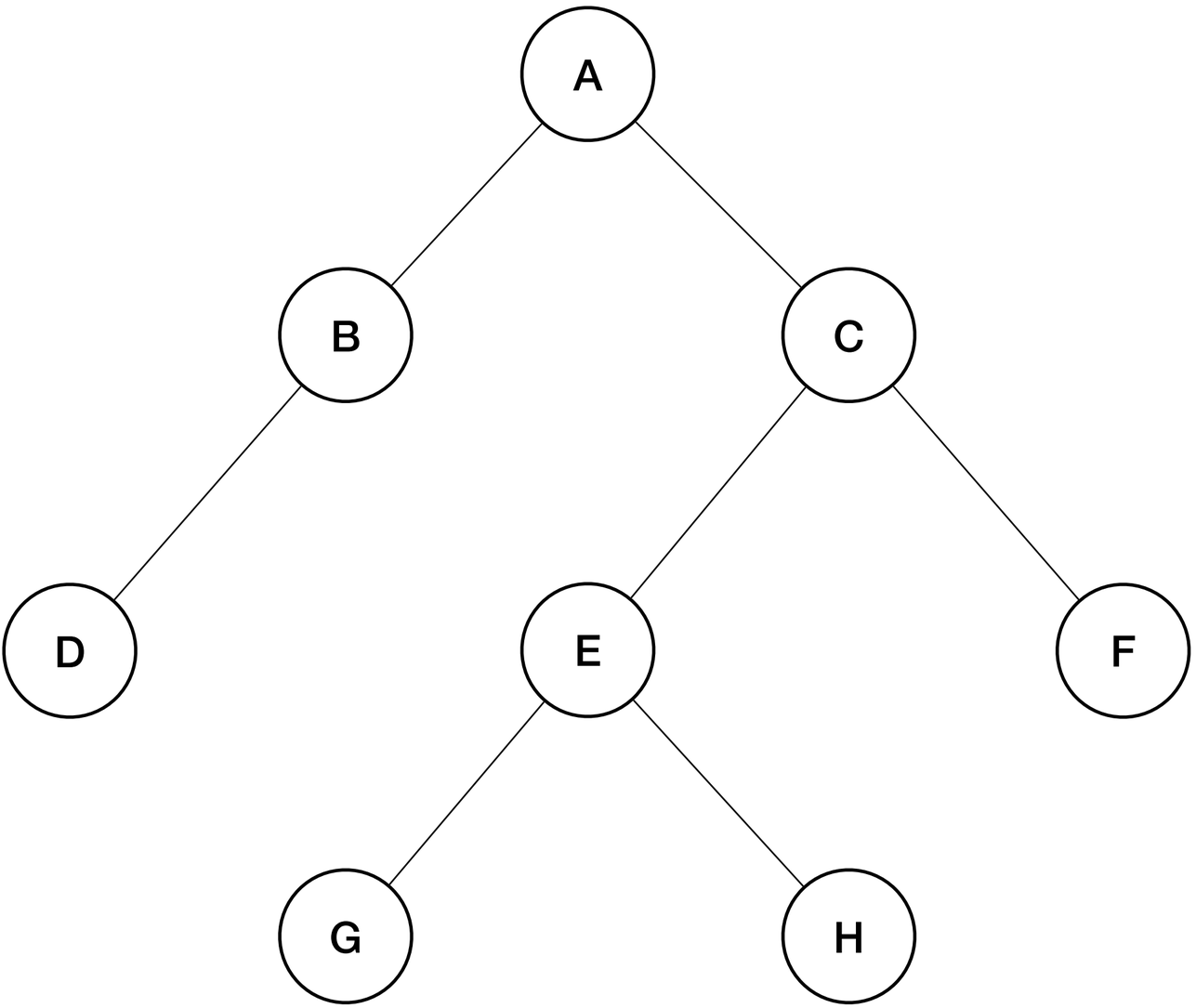
A → B → D → C → E → G → H → F

For each node, visit the **left** subtree, then read the data of the **node**, then visit the **right** subtree.

D → B → A → G → E → H → C → F

For each node, visit the **left** subtree, then visit the **right** subtree, then read the data of the **node**.

Traversing trees



For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

A → B → D → C → E → G → H → F

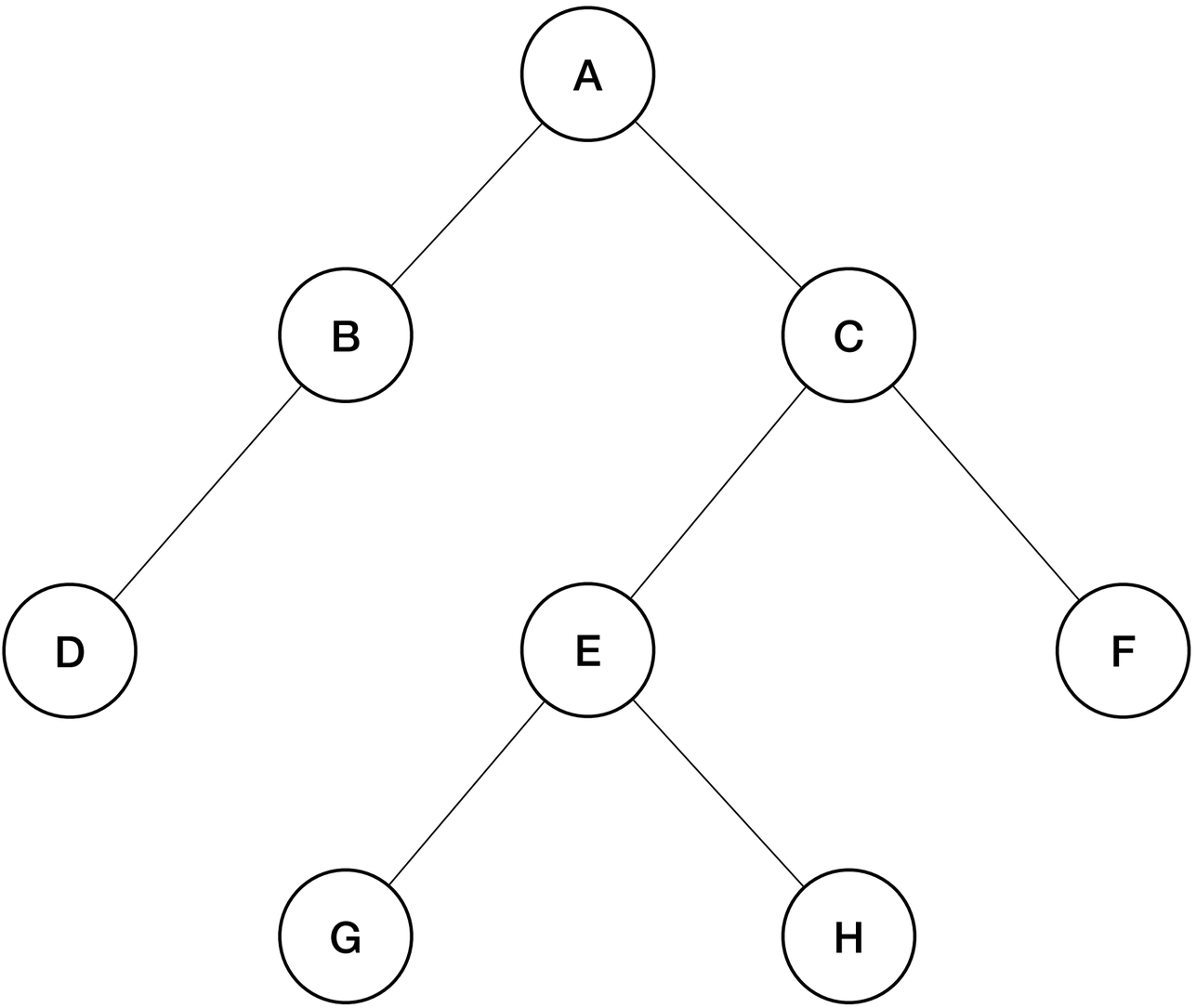
For each node, visit the **left** subtree, then read the data of the **node**, then visit the **right** subtree.

D → B → A → G → E → H → C → F

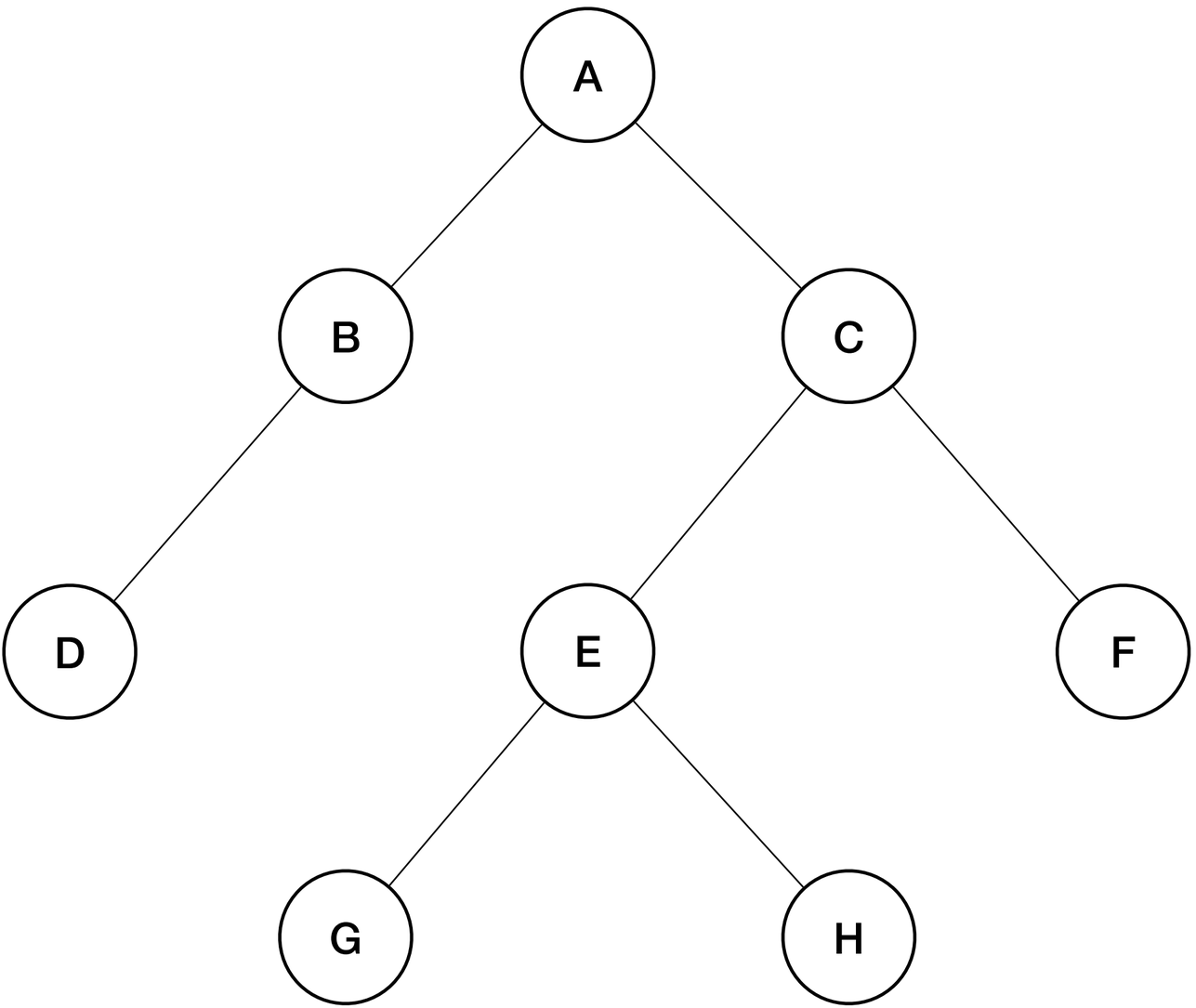
For each node, visit the **left** subtree, then visit the **right** subtree, then read the data of the **node**.

D → B → G → H → E → F → C → A

Traversing trees

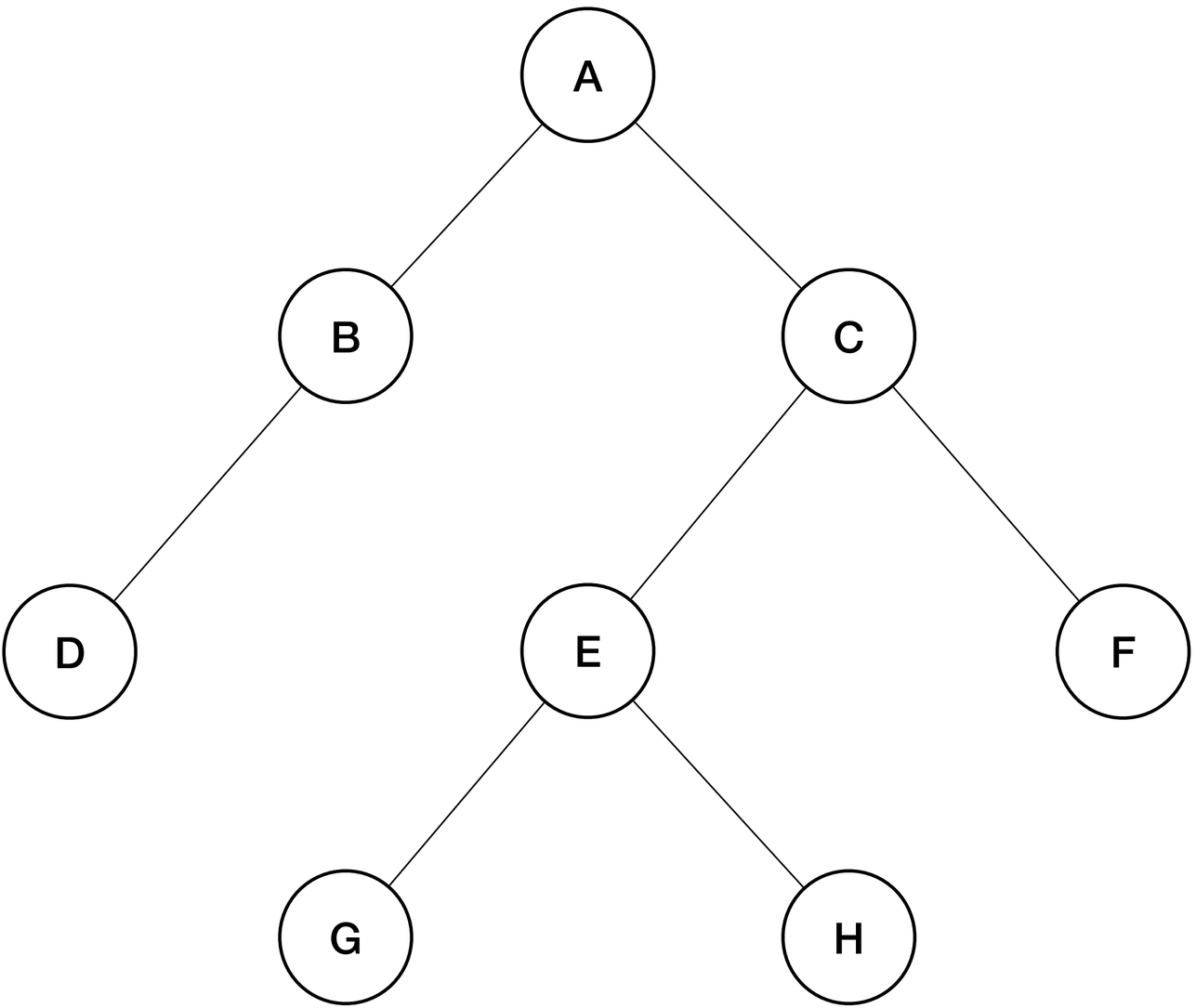


Traversing trees



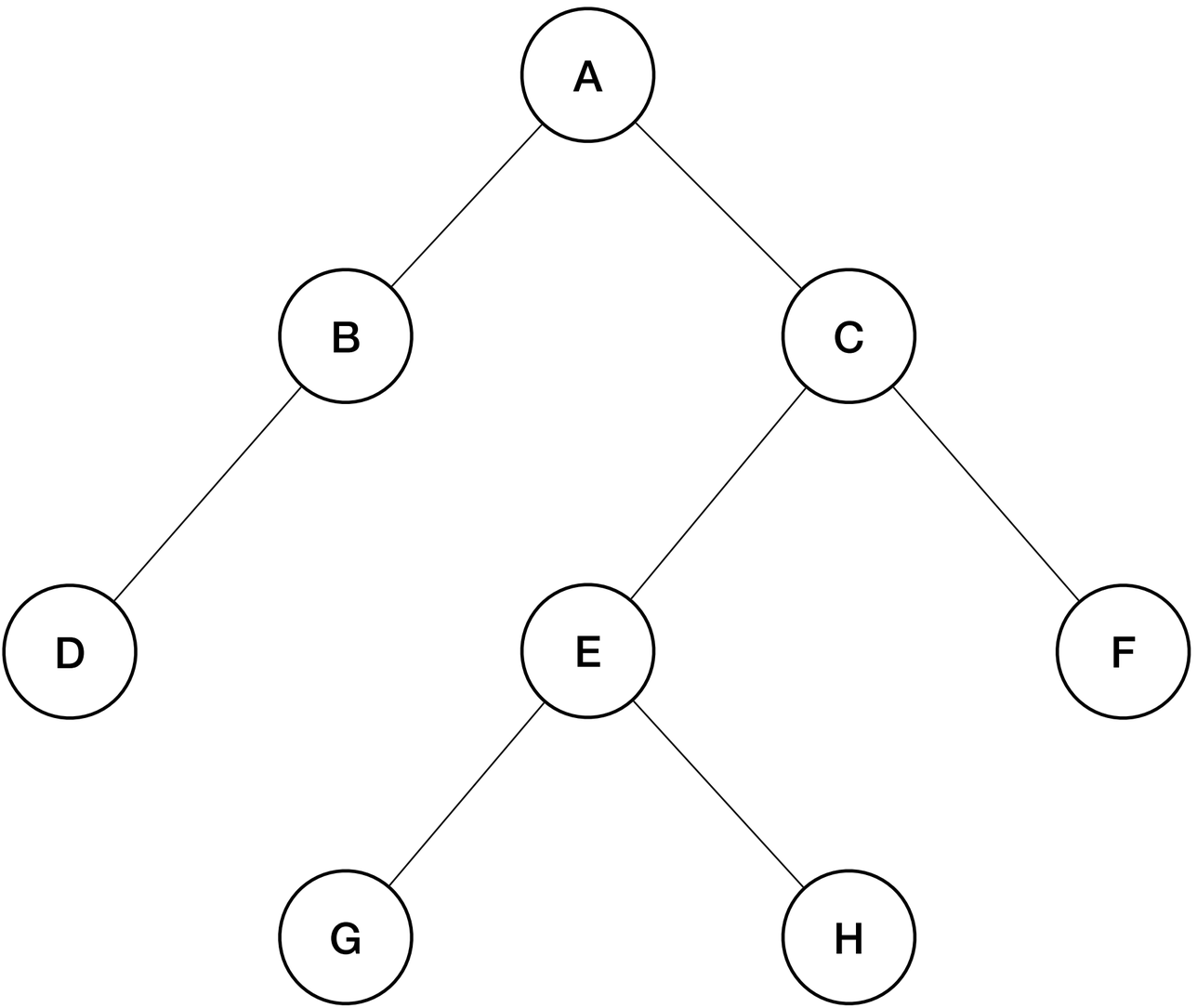
- The previous are called **depth-first** traversals. Could also do a **breadth-first** traversal.

Traversing trees



- The previous are called **depth-first** traversals. Could also do a **breadth-first** traversal.
  - Traverse through all the children of a node, then visit the grandchildren.

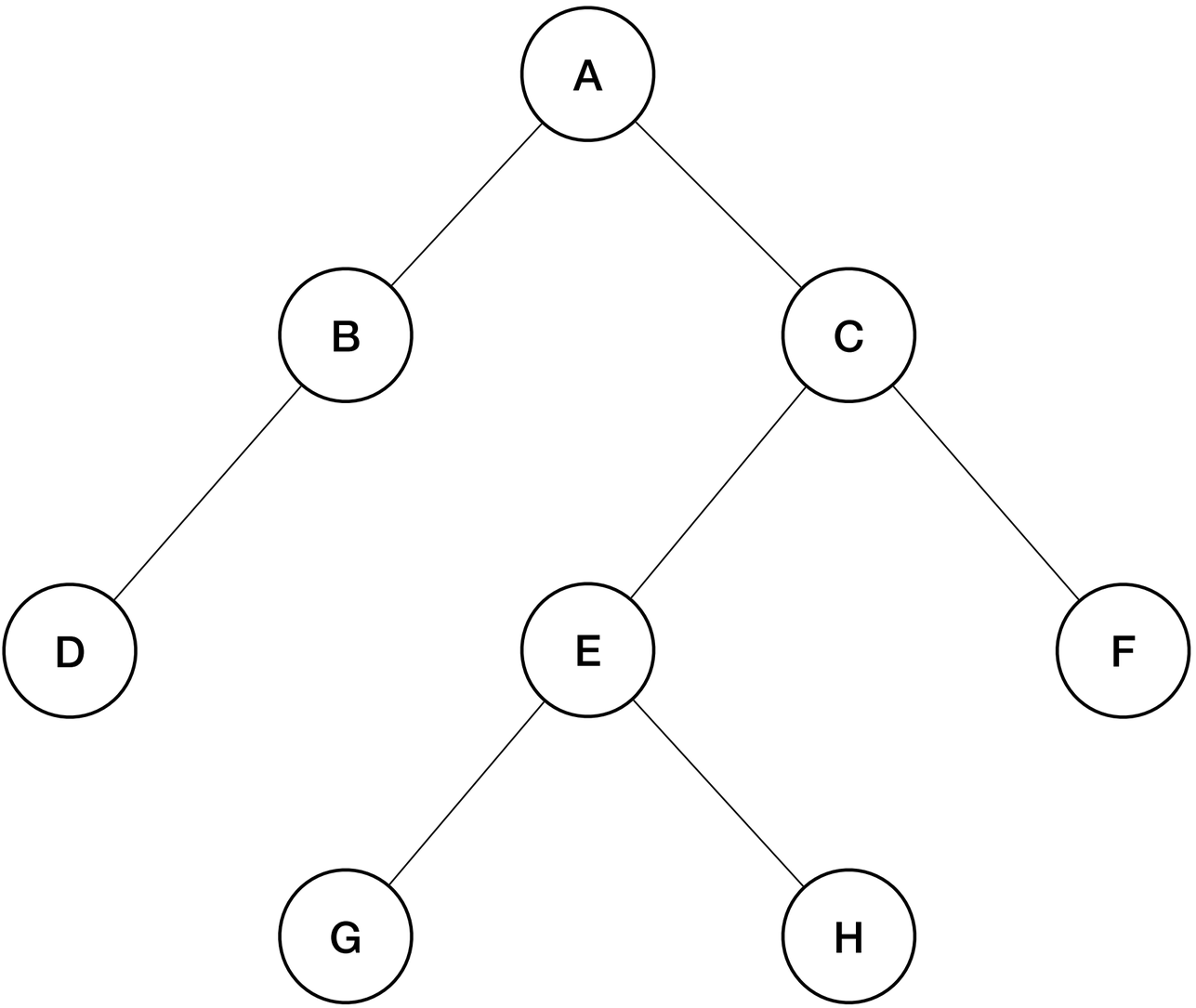
Traversing trees



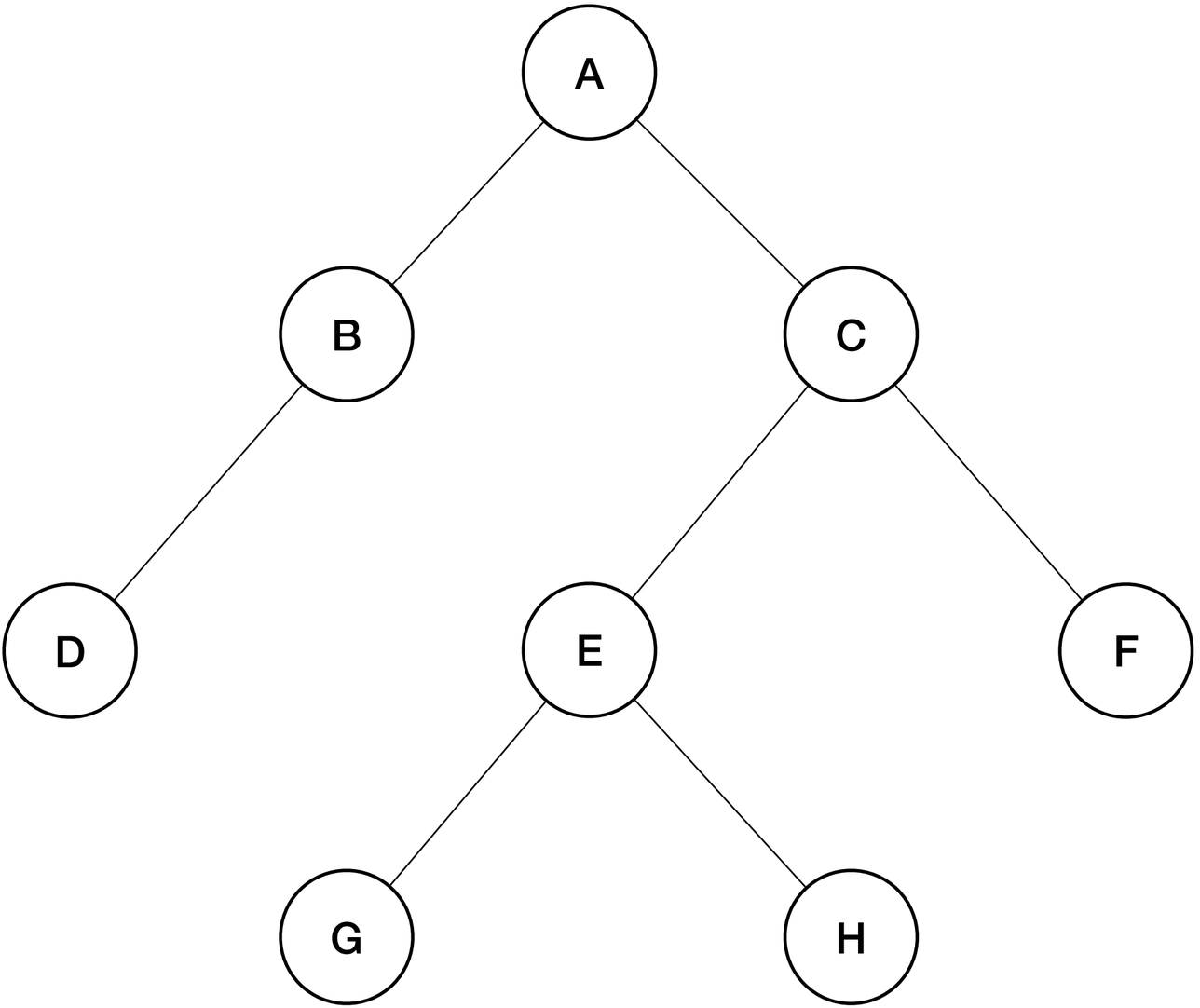
- The previous are called **depth-first** traversals. Could also do a **breadth-first** traversal.
  - Traverse through all the children of a node, then visit the grandchildren.

A → B → C → D → E → F → G → H

Implementing traversals

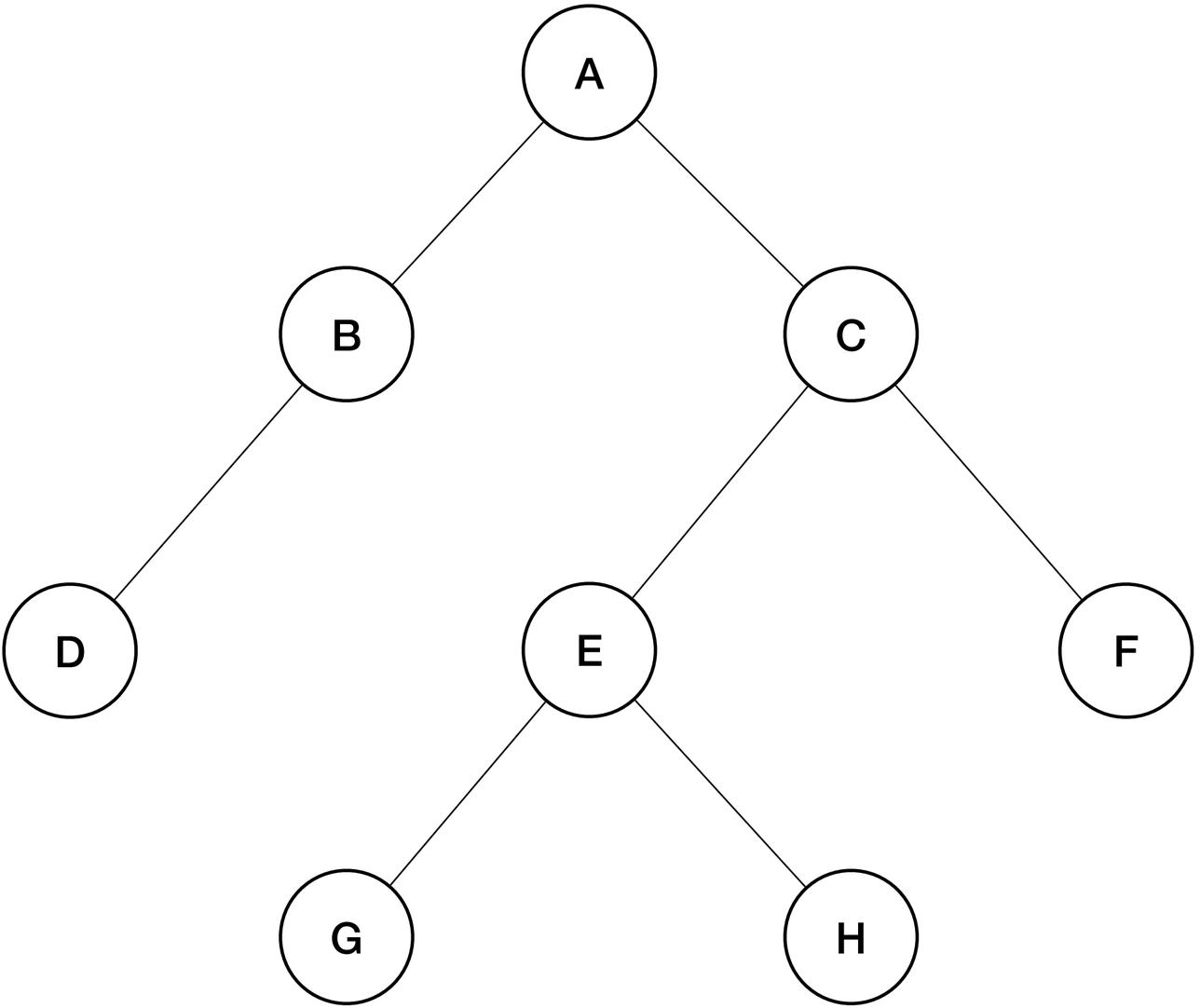


Implementing traversals




```
Queue myQueue
myQueue.enqueue(root)
while(myQueue){
    cursor = myQueue.dequeue()
    cursor.print()
    if (cursor->left)
     myQueue.enqueue(cursor->left)
    if (cursor->right)
     myQueue.enqueue(cursor->right)
}
```

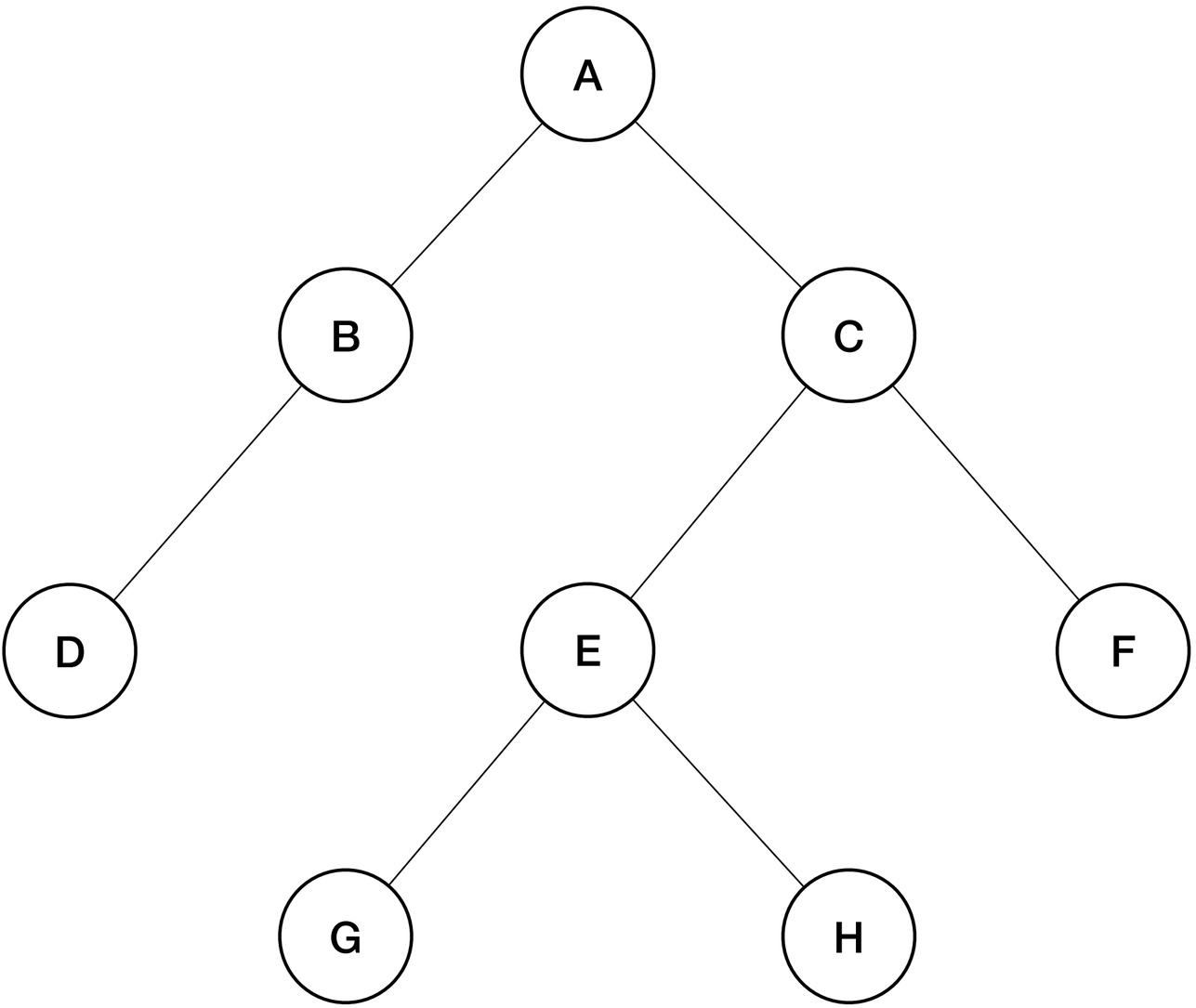
Implementing traversals



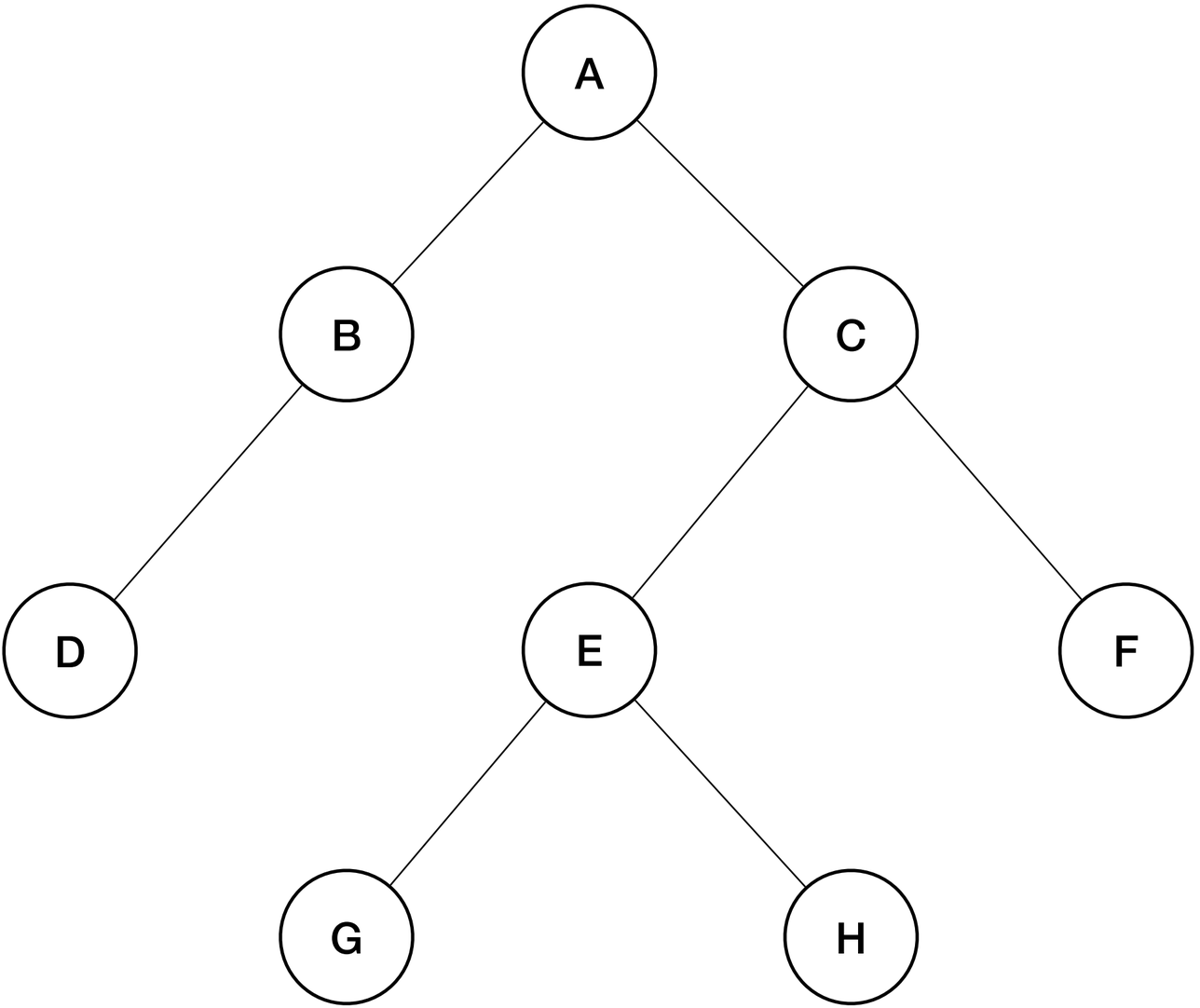

```
Queue myQueue
myQueue.enqueue(root)
while(myQueue){
    cursor = myQueue.dequeue()
    cursor.print()
    if (cursor->left)
        myQueue.enqueue(cursor->left)
    if (cursor->right)
        myQueue.enqueue(cursor->right)
}
```

What does this algorithm do?

Implementing traversals

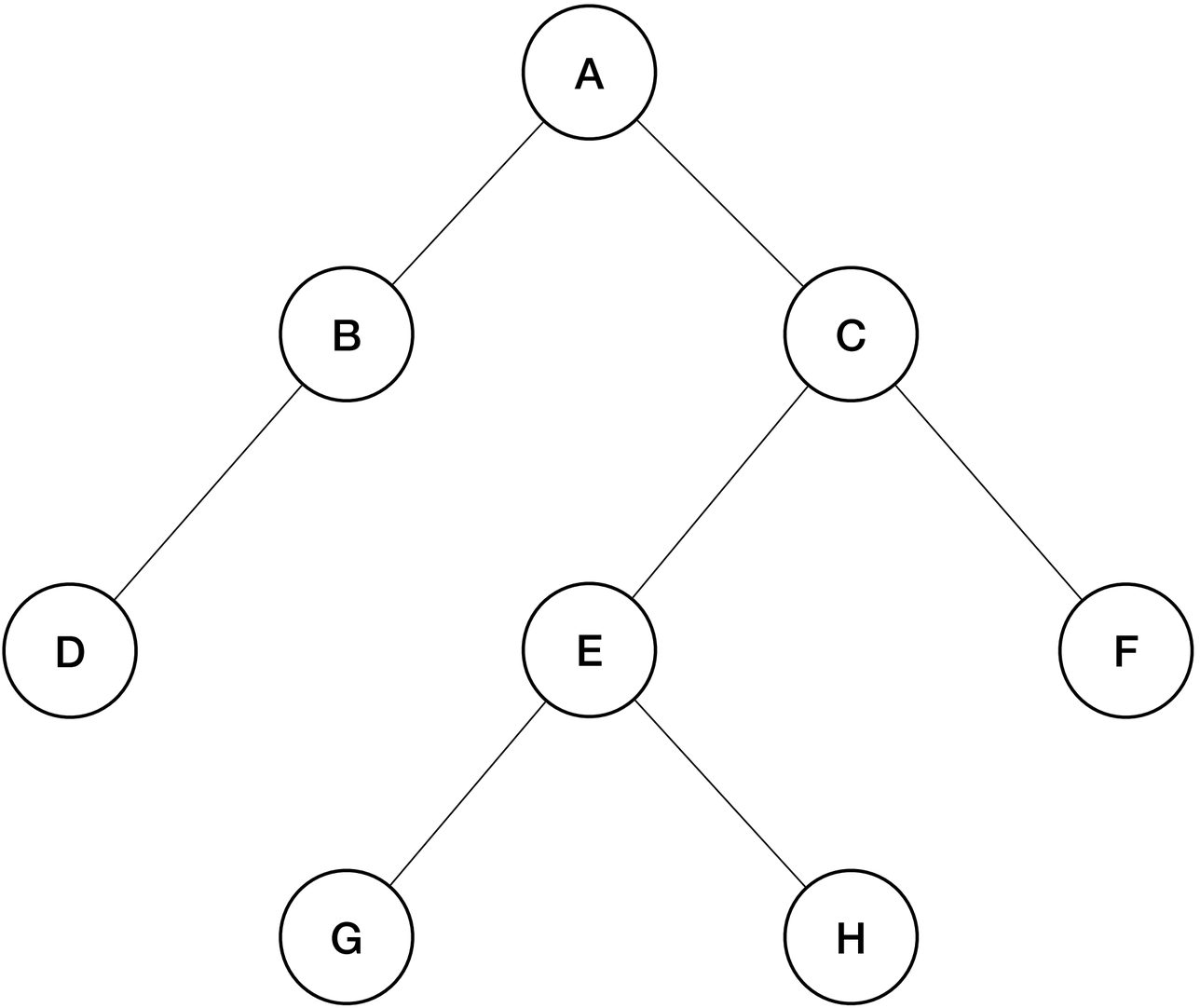


Implementing traversals




```
Stack myStack
myStack.push(root)
while(myStack){
     cursor = myStack.pop()
     cursor.print()
     if (cursor->right)
          myStack.push(cursor->right)
     if (cursor->left)
          myStack.push(cursor->left)
}
```

Implementing traversals




```
Stack myStack
myStack.push(root)
while(myStack){
    cursor = myStack.pop()
    cursor.print()
    if (cursor->right)
        myStack.push(cursor->right)
    if (cursor->left)
        myStack.push(cursor->left)
}
```

What does this algorithm do?

```
typedef struct node{
    int data;
    struct node *left;
    struct node *right;
} node;
```

Practice time

```c
typedef struct node{
    int data;
    struct node *left;
    struct node *right;
} node;
```

Practice time

```c
int main(){
  int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  node * root = (node *) malloc(sizeof(node));
  root->data = arr[0];
  root->left = NULL;
  root->right=NULL;
  node * cursor = root;

  for (int j=0, i=1; i<11; i=i+2, j++){
    add_left(&cursor, arr[i]);
    add_right(&cursor, arr[i+1]);
    cursor = (j%2==0) ? cursor->right : cursor->left;
  }
  print_preorder(root);
  print_inorder(root);
  print_postorder(root);
  delete_tree(root);
}
```

```
typedef struct node{
    int data;
    struct node *left;
    struct node *right;
} node;
```

Practice time

- Write functions to:

```
int main(){
  int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  node * root = (node *) malloc(sizeof(node));
  root->data = arr[0];
  root->left = NULL;
  root->right=NULL;
  node * cursor = root;

  for (int j=0, i=1; i<11; i=i+2, j++){
    add_left(&cursor, arr[i]);
    add_right(&cursor, arr[i+1]);
    cursor = (j%2==0) ? cursor->right : cursor->left;
  }
  print_preorder(root);
  print_inorder(root);
  print_postorder(root);
  delete_tree(root);
}
```

```
typedef struct node{
    int data;
    struct node *left;
    struct node *right;
} node;
```

Practice time

- Write functions to:
  - Add to the left and right of a node.

```
int main(){
  int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  node * root = (node *) malloc(sizeof(node));
  root->data = arr[0];
  root->left = NULL;
  root->right=NULL;
  node * cursor = root;

  for (int j=0, i=1; i<11; i=i+2, j++){
    add_left(&cursor, arr[i]);
    add_right(&cursor, arr[i+1]);
    cursor = (j%2==0) ? cursor->right : cursor->left;
  }
  print_preorder(root);
  print_inorder(root);
  print_postorder(root);
  delete_tree(root);
}
```

```
typedef struct node{
    int data;
    struct node *left;
    struct node *right;
} node;
```

Practice time

- Write functions to:
  - Add to the left and right of a node.
- Implement preorder, inorder, and postorder traversals.

```
int main(){
  int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  node * root = (node *) malloc(sizeof(node));
  root->data = arr[0];
  root->left = NULL;
  root->right=NULL;
  node * cursor = root;
  for (int j=0, i=1; i<11; i=i+2, j++){
    add_left(&cursor, arr[i]);
    add_right(&cursor, arr[i+1]);
    cursor = (j%2==0) ? cursor->right : cursor->left;
  }
  print_preorder(root);
  print_inorder(root);
  print_postorder(root);
  delete_tree(root);
}
```

```c
typedef struct node{
    int data;
    struct node *left;
    struct node *right;
} node;
```

Practice time

- Write functions to:
  - Add to the left and right of a node.
- Implement preorder, inorder, and postorder traversals.
- Delete a tree.

```c
int main(){
  int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  node * root = (node *) malloc(sizeof(node));
  root->data = arr[0];
  root->left = NULL;
  root->right=NULL;
  node * cursor = root;
  for (int j=0, i=1; i<11; i=i+2, j++){
    add_left(&cursor, arr[i]);
    add_right(&cursor, arr[i+1]);
    cursor = (j%2==0) ? cursor->right : cursor->left;
  }
  print_preorder(root);
  print_inorder(root);
  print_postorder(root);
  delete_tree(root);
}
```

Printing a tree

Printing a tree

- Can we print a tree in a human readable way?

Printing a tree

- Can we print a tree in a human readable way?
    - Focus on pre-order traversal

Printing a tree

- Can we print a tree in a human readable way?
    - Focus on pre-order traversal
        - Print node, then go left, then go right

Printing a tree

- Can we print a tree in a human readable way?
    - Focus on pre-order traversal
        - Print node, then go left, then go right
- Use *depth* to print right amount of indentation

Printing a tree

- Can we print a tree in a human readable way?
  - Focus on pre-order traversal
    - Print node, then go left, then go right
- Use *depth* to print right amount of indentation

```
void treeprint(node *cursor, int depth){
  if (cursor == NULL)
    return;
  for (int i = 0; i < depth; i++)
    printf(i == depth - 1 ? "|-" : "  ");
  printf("%d\n", cursor->data);
  treeprint(cursor->left, depth + 1);
  treeprint(cursor->right, depth + 1);
}
```

Printing a tree

- Can we print a tree in a human readable way?
  - Focus on pre-order traversal
    - Print node, then go left, then go right
- Use *depth* to print right amount of indentation

```
void treeprint(node *cursor, int depth){
  if (cursor == NULL)
    return;
  for (int i = 0; i < depth; i++)
    printf(i == depth - 1 ? "|-" : "  ");
  printf("%d\n", cursor->data);
  treeprint(cursor->left, depth + 1);
  treeprint(cursor->right, depth + 1);
}
```

Let us check if we got previous slide right …

Next time

Introduction to C++