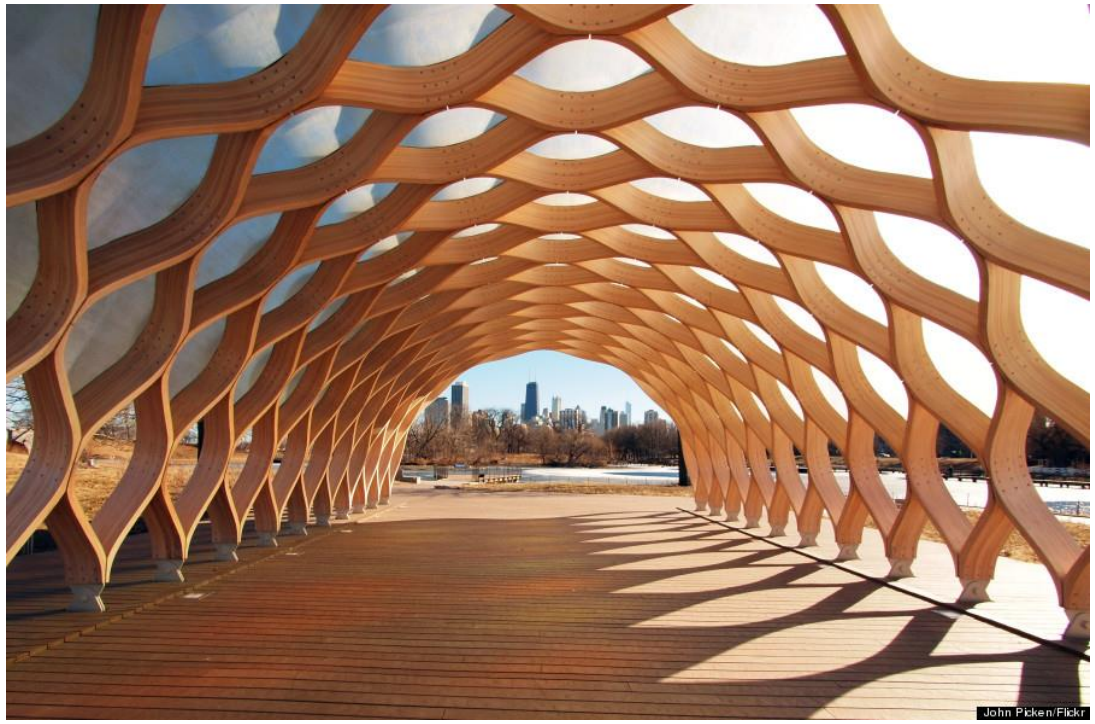
iClicker Attendance

Please click on A if you are here:

A. I am here today.

Data Structures



What is a Data Structure

- A method of organizing (structuring) data to enable problem solving
- We've seen some examples already
  - Arrays (vectors, matrices, and cubes)
  - Structures
- Now that we know about malloc and pointers and structures, we can combine all three and go crazy!

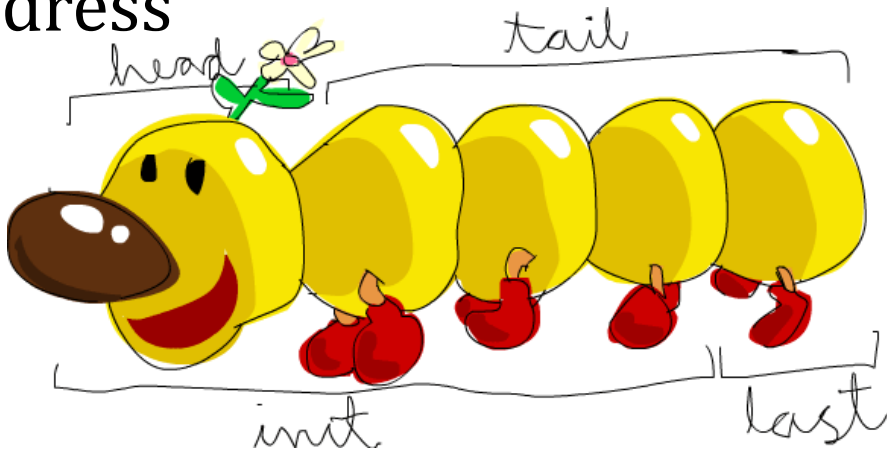
Self Referential Data Structures

- A data structure defines the layout of memory – like a template
- An instance of a data structure contains the memory for the data
- Data structures may contain pointers
- Pointers may be pointers to other INSTANCES of the same layout!
- Violates the "define before use" rule

```
stuct selfref {
        int fld1;
        struct selfref * nextInstancePtr;
}
```

Start simple… singly linked list

- A list has multiple items… each item is a “node”
- Each node carries some data – a payload
  - To keep things simple, our payload will just be a single integer.
  - A more sophisticated application would have a larger and more complex payload
- Each node links to the next node using the address
- Each list has a beginning – a “head” of the list
- Each list has an end – a “tail” of the list



Use a structure to define a node

```
struct sNode {
        int payload;
        struct sNode * next;
};
```

Making new nodes

```
struct sNode * newSnode(int value) {
        struct sNode * newNode;
        newNode=(struct sNode *)malloc(sizeof(struct sNode));
        if (newNode) {
                newNode->payload=value;
                newNode->next=NULL;
        }
        return newNode;
}
```

Head and Tail

- Keep track of the head of the list with a pointer:

```
struct sNode * head=NULL;
```

- The tail of the list is the node whose next node pointer is NULL

- An empty list is a list in which head==NULL

Starting with a single node

```
struct sNode * head = newSnode(13);
…
free(head);
```

head: 0x600010530

Memory @ 0x600010530 [0000000D 0000000000000000]

Graphically:

Inserting a new node at the tail

```
void addSTail(struct sNode * head,int value) {
        assert(head!=NULL);
        while(head->next != NULL) head=head->next;
        head->next=newSnode(value);
}
```

Example multi-node list

```
struct sNode * head=newSnode(13);
addSTail(head,23);
addSTail(head,9);
```

head: 0x600010530

Memory @ 0x600010530 [0000000D 0x6000105a0]

Memory @ 0x6000105a0 [00000017 0x6000105c0]

Memory @ 0x6000105c0 [00000009 0x000000000]

Freeing an Entire List

```
void freeSList(struct sNode *head) {
        if (head==NULL) return;
        freeSList(head->next);
        free(head);
}
```

Recursion and Data Structures

- Reduce the problem by thinking of how to take a SINGLE STEP
  - That makes the problem a little bit simpler
- For instance, to free an entire list, all I have to do is free the sub-list that starts at head->next. When that is done, all that is left is to free the node at that head is pointing to.
  - The sub-list at head->next is smaller than the list at head
  - Therefore, the problem of freeing the sub-list has to be easier than the problem of freeing the whole list
  - If the sub-list is empty, then we're almost done.

Freeing a multi-node list

```
freeSlist(head);
```

head: 0x600010530
Memory @ 0x600010530 [0000000C 0x6000105a0]
Memory @ 0x6000105a0 [00000017 0x6000105c0]
Memory @ 0x6000105c0 [00000009 0x000000000]

Freeing a multi-node list

freeSlist(head); freeSlist(head->next)

head: 0x6000105a0

Memory @ 0x600010530 [0000000C 0x6000105a0]

Memory @ 0x6000105a0 [00000017 0x6000105c0]

Memory @ 0x6000105c0 [00000009 0x000000000]

Freeing a multi-node list

freeSlist(head); freeSlist(head->next); freeList(head->next)

head: 0x6000105c0

Memory @ 0x600010530 [0000000C 0x6000105a0]

Memory @ 0x6000105a0 [00000017 0x6000105c0]

Memory @ 0x6000105c0 [00000009 0x000000000]

Freeing a multi-node list

freeSlist(head); freeSlist(head->next); freeList(head->next)
freeSlist(NULL)

head: 0x00000000

Memory @ 0x600010530 [0000000C 0x6000105a0]
Memory @ 0x6000105a0 [00000017 0x6000105c0]
Memory @ 0x6000105c0 [00000009 0x000000000]

Freeing a multi-node list

freeSlist(head); freeSlist(head->next); freeList(head->next)
free(head)

head: 0x6000105c0

Memory @ 0x600010530 [0000000C 0x6000105a0]

Memory @ 0x6000105a0 [00000017 0x6000105c0]

Memory @ 0x6000105c0 [00000009 0x000000000]

Freeing a multi-node list

freeSlist(head); freeSlist(head->next)
free(head)

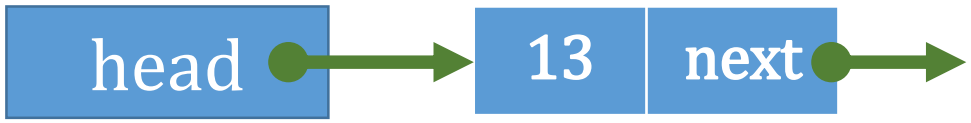
head: 0x6000105a0

Memory @ 0x600010530 [0000000C 0x6000105a0]

Memory @ 0x6000105a0 [00000017 0x6000105c0]

Freeing a multi-node list

```
freeSlist(head);
free(head);
```



head: 0x600010530

Memory @ 0x600010530 [0000000C 0x6000105a0]

Adding at the front of the list

```
struct sNode * addSHead(struct sNode * head, int value) {
        struct sNode * new=newSnode(value);
        new->next=head;
        return new;
}
```

Example front loaded list

```
struct sNode * head=newSnode(13);
head=addSHead(head,23);
head=addSHead(head,9);
```

head: 0x6000105c0

Memory @ 0x6000105c0 [00000009 0x6000105a0]

Memory @ 0x6000105a0 [00000017 0x600010530]

Memory @ 0x600010530 [0000000D 0x000000000]

Adding in Numeric Order

```
struct sNode * addSOrdered(struct sNode * head, int value) {
        if (head==NULL) return newSnode(value);
        if (value<head->payload) {
                struct sNode * new = newSnode(value);
                new->next=head;
                return new;
        }
        head->next=addSOrdered(head->next,value);
        return head;
}
```

iClicker Question

- What is the recursive step?

```
if (head==NULL) return newSnode(value);
if (value<head->payload) {
        struct sNode * new = newSnode(value);
        new->next=head;
        return new; }
head->next=addSOrdered(head->next,value);
```

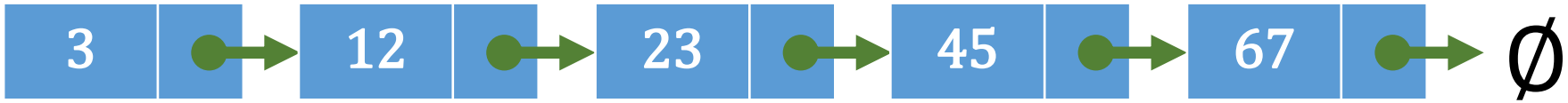
A. if value<head add to front, else add to next sublist
B. if value>head add to front, else add to next sublist
C. if value<head add to tail, else add to next sublist
D. if value>head add to tail, else add to next sublist
E. None of the above

Making a sorted list

see xmp_sll

```
head=NULL; int i;
for(i=1;i<argc; i++) {
        head=addSOrdered(head,atoi(argv[i]));
}
```

./sll 12 45 67 23 3

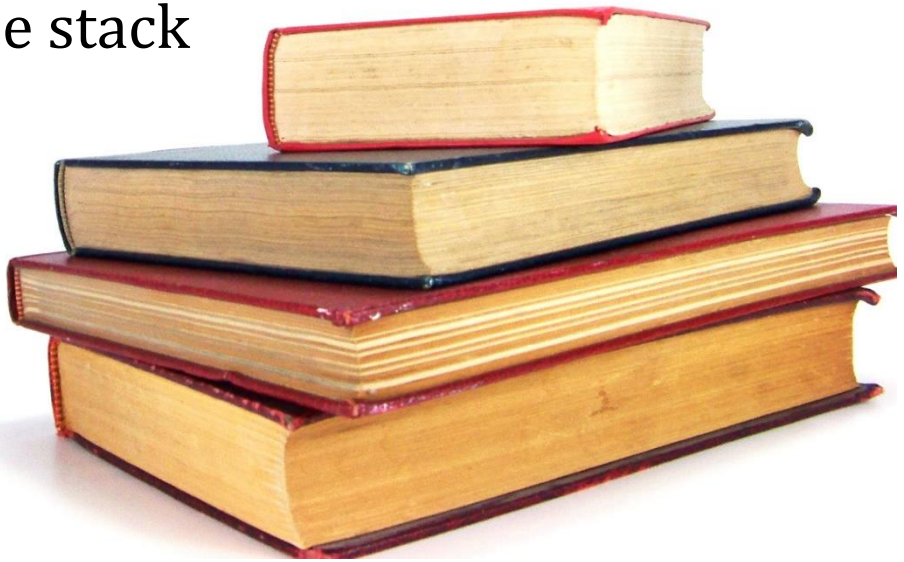


Stacks

- A stack is a data structure that holds multiple elements
- A stack supports two data operations… push and pop

Activation Records are kept in a stack

- "Current" function on top of stack
- When a function is invoked,
  - add its activation record to the top of the stack
- After a function returns,
  - remove its activation record from the top of the stack



Singly Linked Lists are good for stacks!

- Push and pop from the head of the stack

```
void push(struct sNode **stack,int value) {
   if ((*stack)==NULL)
      (*stack)=newSnode(value);
   else (*stack)=addSHead((*stack),value);
}

int more(struct sNode **stack) {
   return (*stack)!=NULL;
}
```

```
int pop(struct sNode **stack) {
        assert((*stack)!=NULL);
        struct sNode * node=(*stack);
        int pval=node->payload;
        (*stack)=node->next;
        free(node);
        return pval;
}
```

Using a Stack

```
int main(int argc,char **argv) {
        int i;
        struct sNode * stack=NULL;
        for(i=1;i<argc; i++) push(&stack,atoi(argv[i]));
        printf("Args in reverse... ");
        while(more(&stack))  printf("%d ",pop(&stack));
        printf("\n");
        return 0;
}
```

Doubly Linked List

- Like two linked lists in one…
  - Singly Linked List: head -> next -> next -> … -> null
  - Singly Linked List: tail -> prev -> prev -> … -> null
- Each node points to its previous neighbor and its next neighbor
  - When we modify lists, remember to change both prev and next links
- The entire list has both a head pointer and a tail pointer
- When you change a doubly linked list, you might change:
  - head pointer
  - tail pointer
  - both head and tail pointer
  - neither head nor tail pointer

Doubly Linked Lists

```
struct dNode {
        struct dNode * prev;
        int payload;
        struct dNode * next;
};

struct dList {
        struct dNode * head;
        struct dNode * tail;
};
```

Why doubly linked lists?

- Fast read from front or back of the list
  - Forward or Reverse without pre-traversing the list
- Fast insert at front or back of the list
  - insert without traversing the list
- Fast navigation to neighbors in both directions
  - No need to traverse the entire list to find your predecessor

- But…
  - Extra memory for two pointers per node
  - Extra processing to maintain both sets of pointers

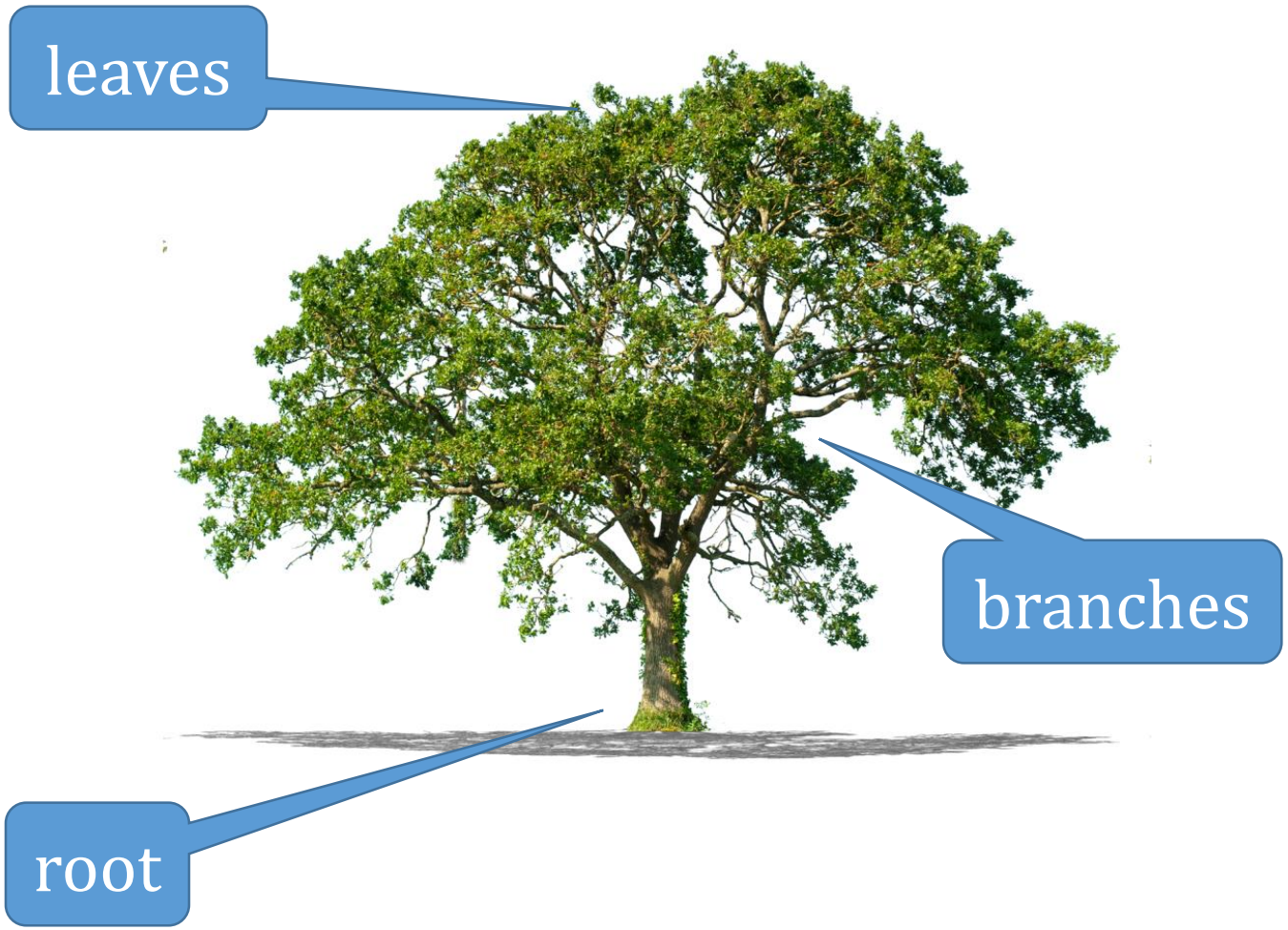
Adding to the head of a Doubly Linked

```
void addDHead(struct dList * list, int value) {
        struct dNode * new = newDnode(value);
        if (list->head==NULL) {
                assert(list->tail==NULL);
                list->head=list->tail=new;
                return;
        }
        list->head->prev=new;
        new->next=list->head;
        list->head=new;
}
```

Adding to a DLL in order

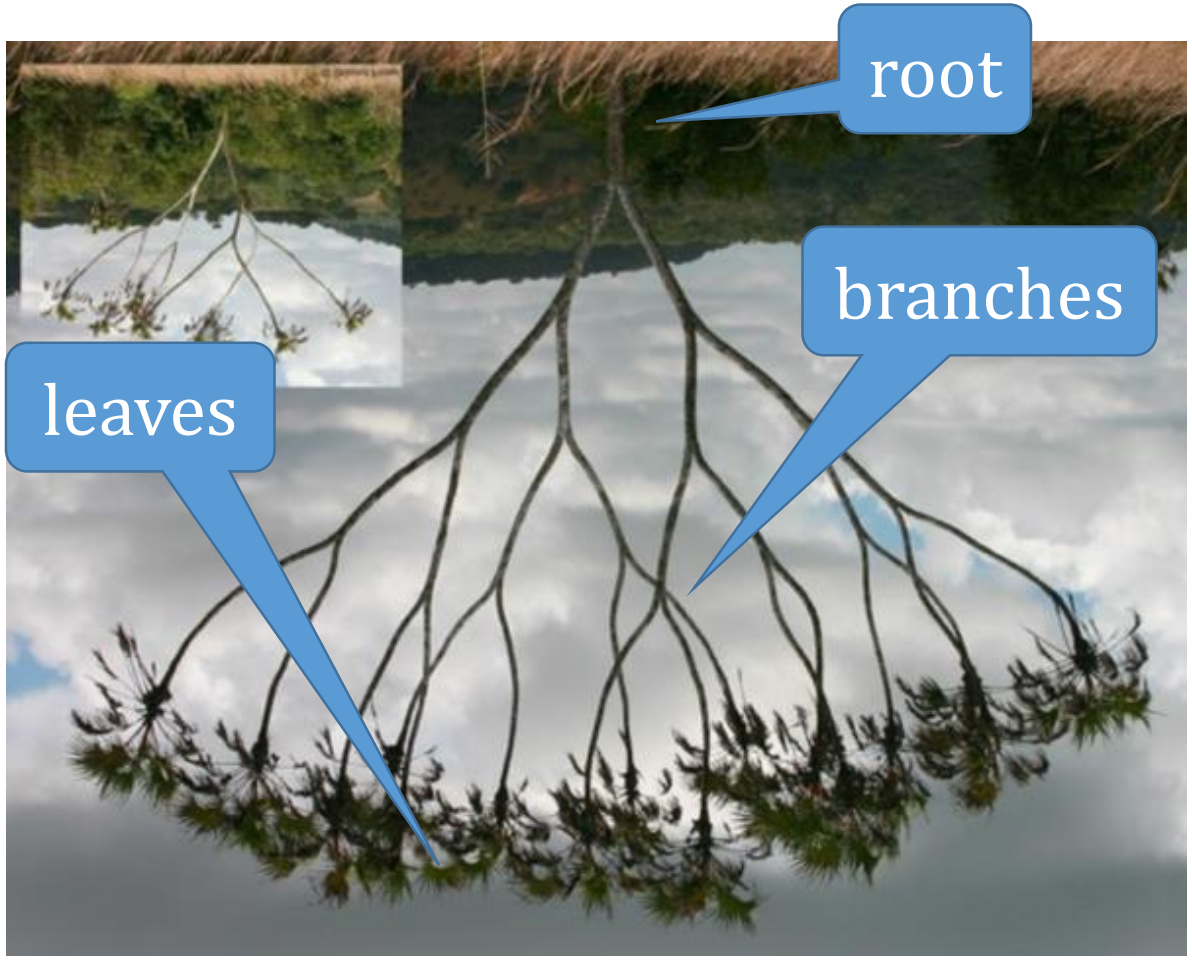
```
void addDOrdered(struct dList * list, int value) {
    if (list->head==NULL ||
        list->head->payload>value) {
        addDHead(list,value);
        return;
    }
    struct dNode * node;
    for(node=list->head;
        node!=NULL && node->payload<value;
        node=node->next) {}
    if (node==NULL) {
        addDTail(list,value);
        return;
    }
    // node-> first node whose value is >= value
    struct dNode * new = newDnode(value);
    new->prev=node->prev;
    node->prev=new;
    new->prev->next=new;
    new->next=node;
}
```

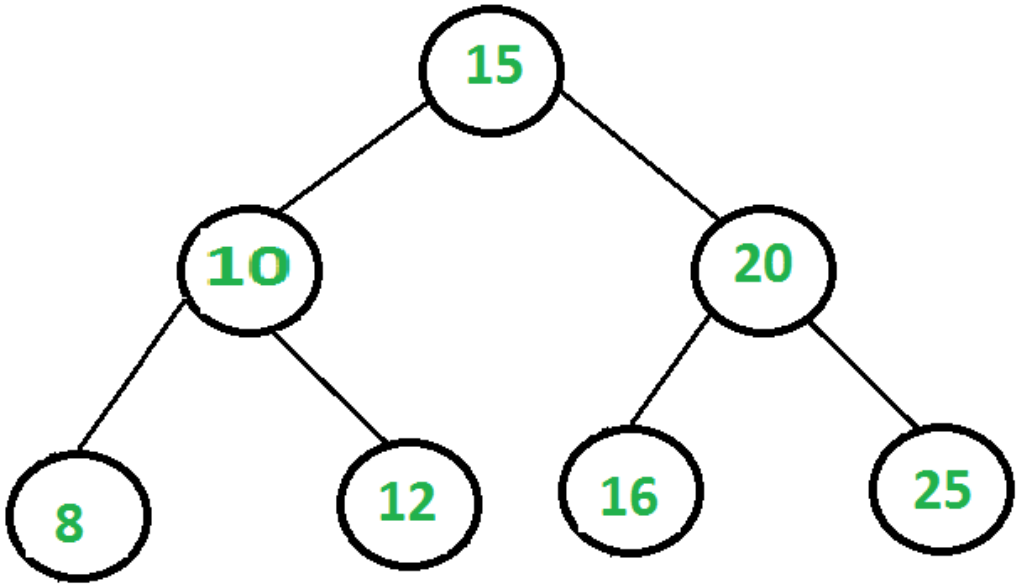
Tree



Binary Tree

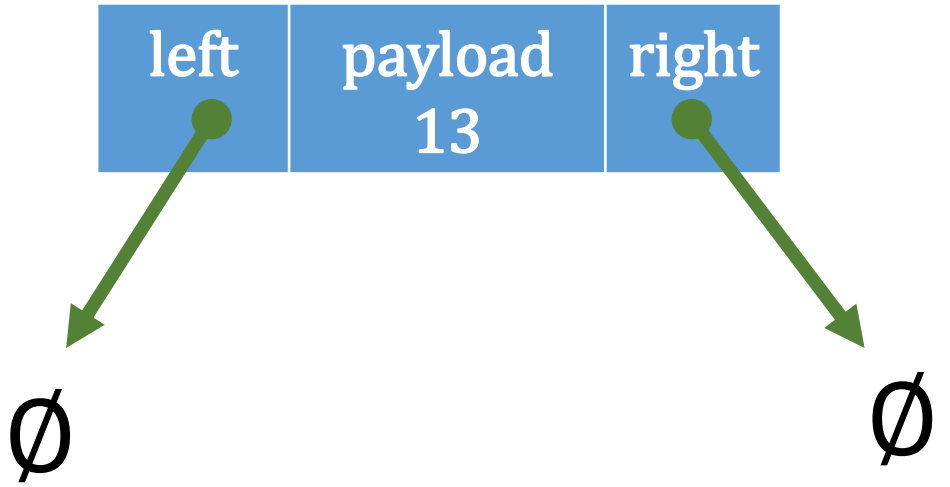
Computer Science Binary Tree





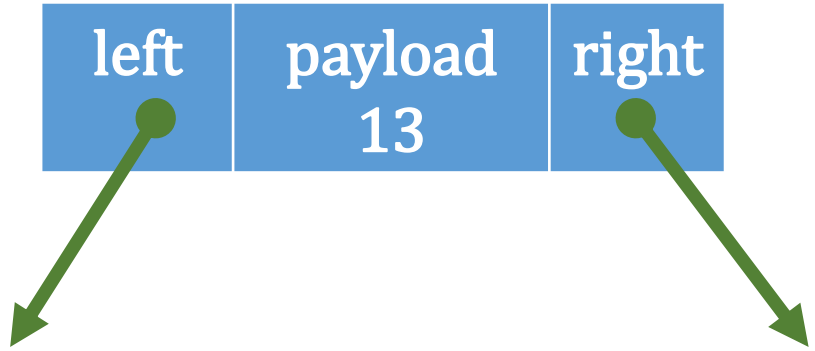
Tree Node Structure

```
struct bTree {
        struct bTree * left;
        int payload;
        struct bTree * right;
};
```



Single Node Tree

```
struct bTree * root=newBTree(13);
```



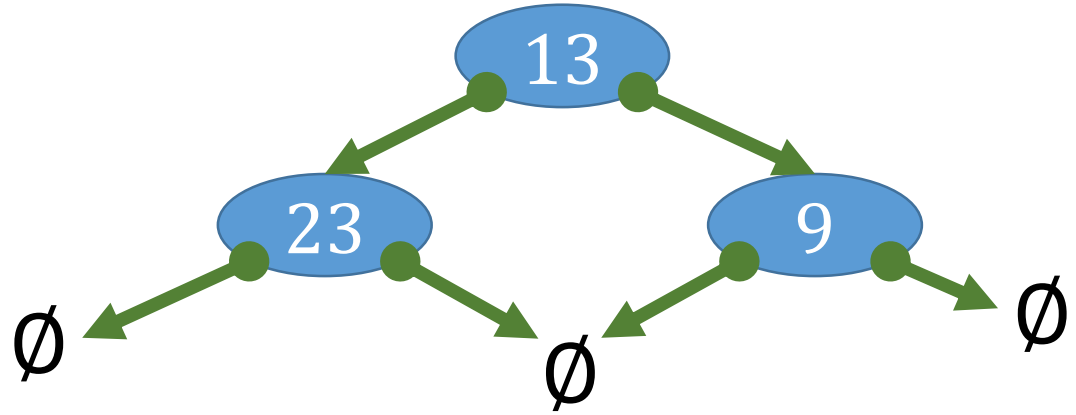
@ 0x600010530 { payload=13 left=0x0 right=0x0 }

Example 3 node tree

@ 0x600010570 { payload=13 left=0x600010590 right=0x6000105b0 }

@ 0x600010590 { payload=23 left=0x0 right=0x0 }

@ 0x6000105b0 { payload=9 left=0x0 right=0x0 }



Trees are Wonderfully Recursive!

- root -> left is the left sub-tree
  - It may or may not be empty
- root -> right is the right sub-tree
  - It may or may not be empty

```
int sumTree(struct bTree * root) {
        int sum=root->payload;
        if (root->left) sum+=sumTree(root->left);
        if (root->right) sum+=sumTree(root->right);
        return sum;
}
```

Sorted Binary Trees

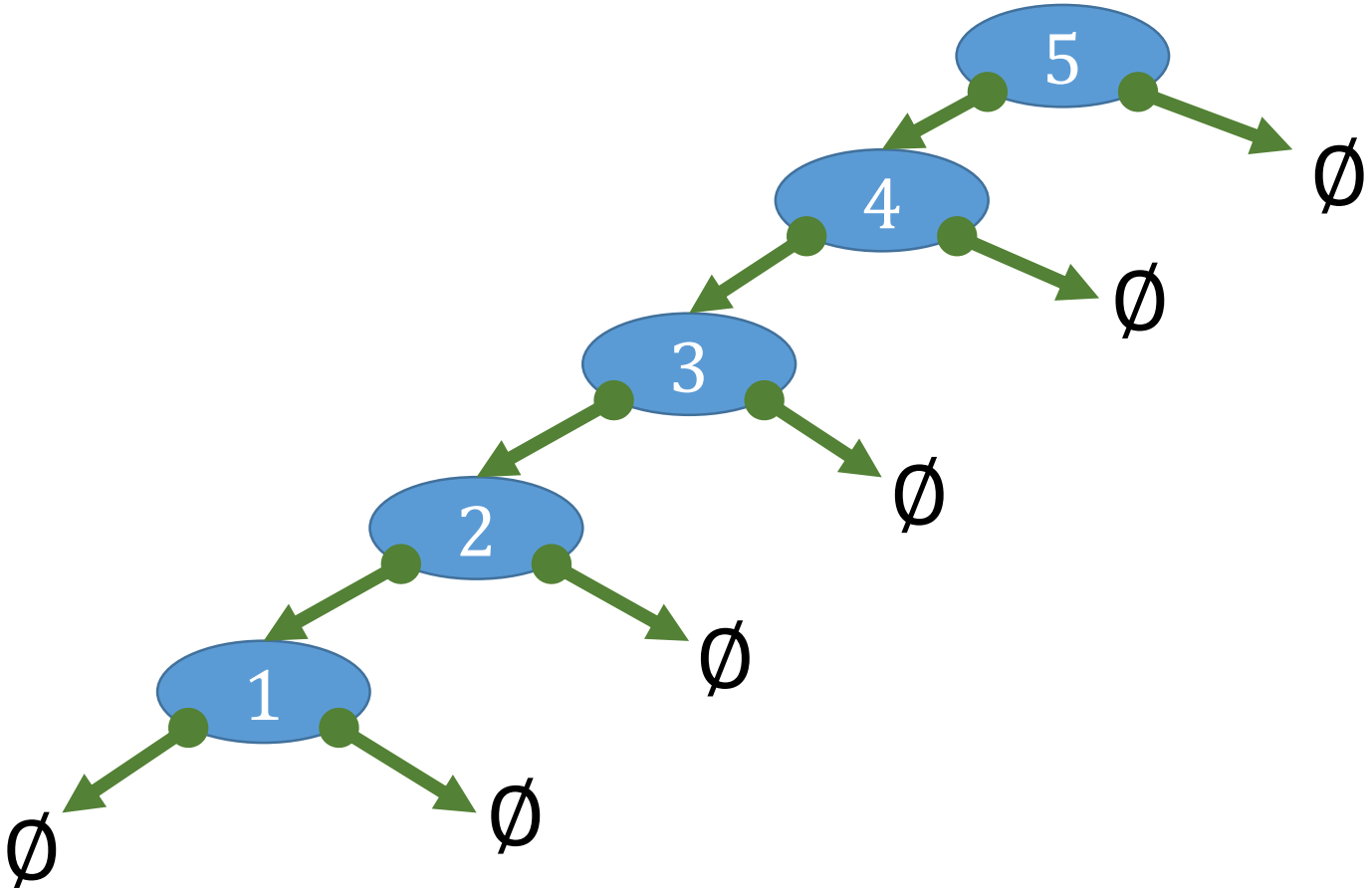
- Binary trees with a special property…
- Everything in the left sub-tree is less than or equal to the payload
- Everything in the right sub-tree is greater than the payload

```
void printIncreasing(struct bTree *root) {
        if (root->left) printIncreasing(root->left);
        printf("%d\n",root->payload);
        if (root->right) printIncreasing(root->right);
}
```

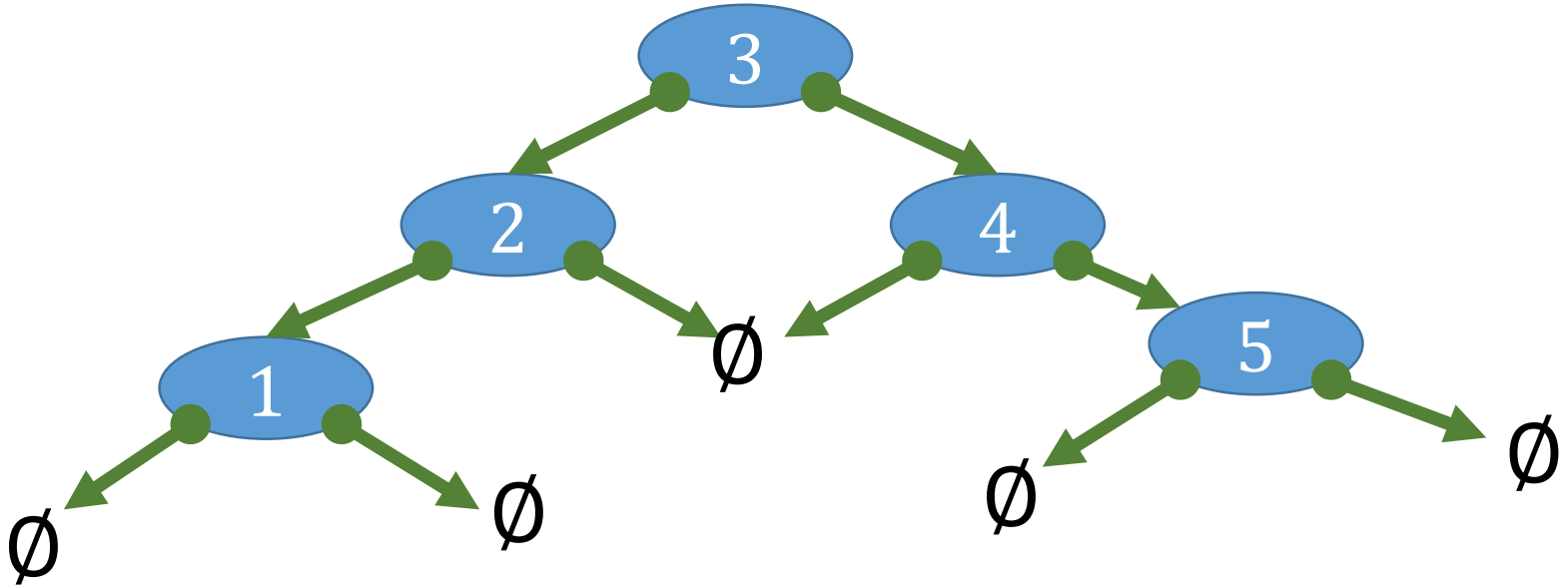
Adding to sorted binary trees

```
void addBtree(struct bTree * root, int value) {
   assert(root!=NULL);
   if (value<root->payload) {
      if (root->left==NULL) root->left=newBtree(value);
      else addBtree(root->left,value);
   } else {
      if (root->right==NULL) root->right=newBtree(value);
      else addBtree(root->right,value);
   }
}
```

Unbalanced Binary Trees



(More) Balanced Binary Tree



Binary Tree Depth

- The depth of a binary tree is the distance from the root to the farthest leaf node

```
int depthBTree(struct bTree *root) {
        if (root==NULL) return 0;
        int dleft=depthBTree(root->left);
        int dright=depthBTree(root->right);
        return 1+((dleft > dright) ? dleft : dright);
}
```

What depth can I expect?

- Best case – completely balanced binary tree , Each level (except the last) is completely full
  - First level has one node
  - Second level has two nodes
  - Third level has four nodes
  - …
  - m^{th} level has <= $2^{(m-1)}$ nodes
- $nodes \leq \sum_{d=1}^{m-1} 2^{d-1} = 2^m - 1 < 2^m$
- Or, $m \geq \log_2 nodes$

Why do I care about depth?

- Time to insert is proportional to the depth
- Time to find a new entry is proportional to depth
- If I have n data items, and it takes $\log_2 n$ time to insert each data item, it takes about $n \times \log_2 n$ time to insert all n data items

- Sort time is $n \log_2 n$

Resources

- Wikipedia: Linked List, Trees, B-Tree