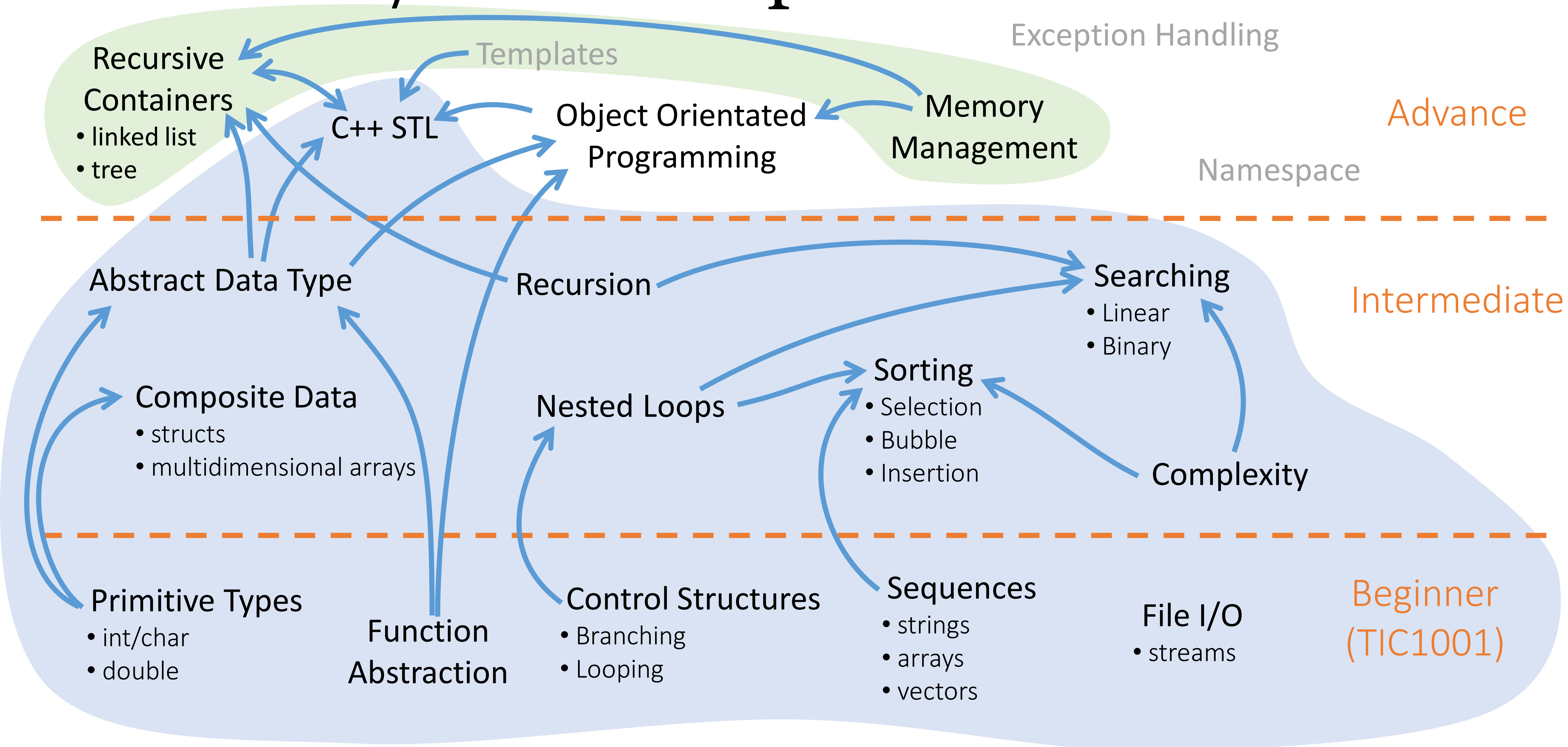
Lecture 7
Memory State and Recursive Containers

TIC1002 Introduction to Computing and Programming II

TIC1001/2 Roadmap



Previously

Abstract Data Types

– Data + Operations

Power of Abstraction

– Implementation is hidden
– Only need to know the specification to use

Homemade ADT

– Stack, Queue
– Two ways to implement Queue

1. Using vector
2. Using two stacks

- Pros and cons of each implementation

C++ STL

– Vector, Map & Set

Quick Primer on Templates

C++ requires all variables and fields to have a type

- Compiler knows how much memory to allocate
- Type is statically determined at compile time

```
struct Point {
    double x, y;
};
```

Becomes a problem when you have containers

```
struct Box {
    int item;
};
```

- Box can only contain int type
- Containers should be generic to be useful
- Cannot determine the type of its contents at compile time

Templates as Placeholders

Specify arbitrary types as placeholders

```
template <typename T>
struct Box {
    T item;
};
```

- Now our Box can contain item of type T
- The exact type of that T represents is specified when Box is declared

```
Box<int> int_box;
Box<string> str_box;
Box<Point> pt_box;

int_box.item = 1;
str_box.item = "a string";

Point p;
pt_box.item = p;
```

Templates as Placeholers

Within the declaration

– Placeholder refers to a constant type

```
template <typename T>
struct Pair {
    T first;
    T second;
};
```

– first and second fields must be the same type

– Can have multiple placeholders

```
template <typename K,
          typename V>
struct Pair {
    K first;
    V second:
}
```

– first and second fields can be of different types

The C++ Memory Model

Understanding the Memory State

All computation processes requires memory

- To store and operate on variables
- Operating System will allocate some memory to a process

In C/C++, memory is organized into

1. Stack memory

- Statically allocated by compiler
- So far, this is the only memory we have used

2. Heap memory

- Dynamically allocated by code

Primitives are Values

What's the output?

```
int x = 10;
int y = x;
x = 5;
cout << x << "," << y << endl;
```

Values are copied

```
int &z = x;
x = 42;
cout << x << "," << z << endl;
```

References are aliased

https://goo.gl/6qKRZb

Swapping Variables

Basic swap function

```
void swap(int x, int y) {
    int t = x;
    x = y;
    y = t;
}
```

– swap function only works with int variables

Make generic

```
template <typename T>
void swap(T x, T y) {
    T t = x;
    x = y;
    y = t;
}
```

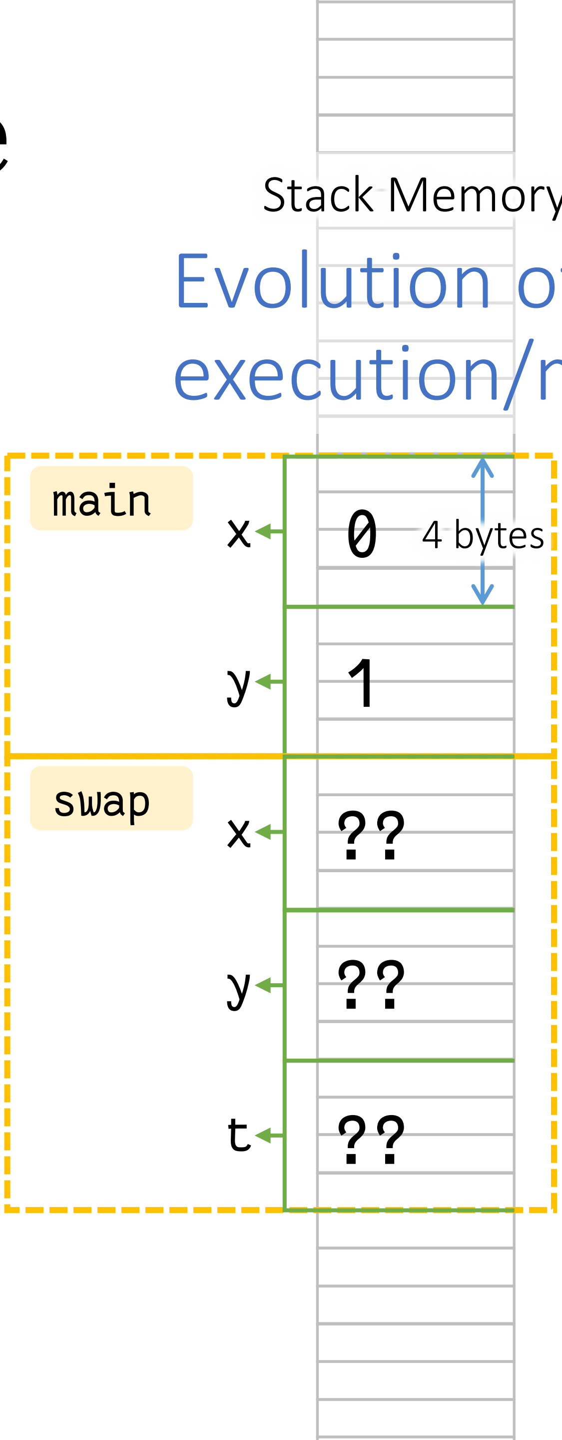
– Swap now works for any two inputs of the same type

C++ Memory State

Back to basics

```
template <typename T>
void swap(T x, T y) {
    T t = x;
    x = y;
    y = t;
}

int main() {
    int x = 0, y = 1;
    swap(x, y);
    cout << x << y << endl;
}
```

Evolution of the execution/memory state



Function call

creates a new stack frame as the execution environment

Variables declared in function are allocated space on the stack

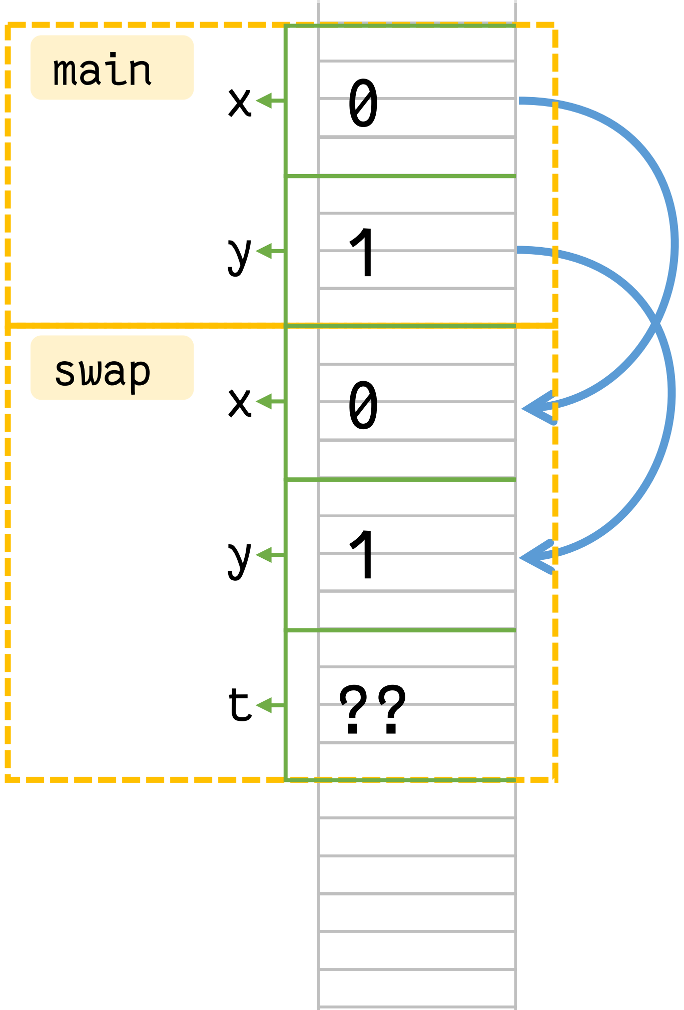
C++ Memory State

What is the output?

```
template <typename T>
void swap(T x, T y) {
    T t = x;
    x = y;
    y = t;
}

int main() {
    int x = 0, y = 1;
    swap(x, y);
    cout << x << y << endl;
}
```

Stack Memory

Evolution of the execution/memory state



Pass-by-value

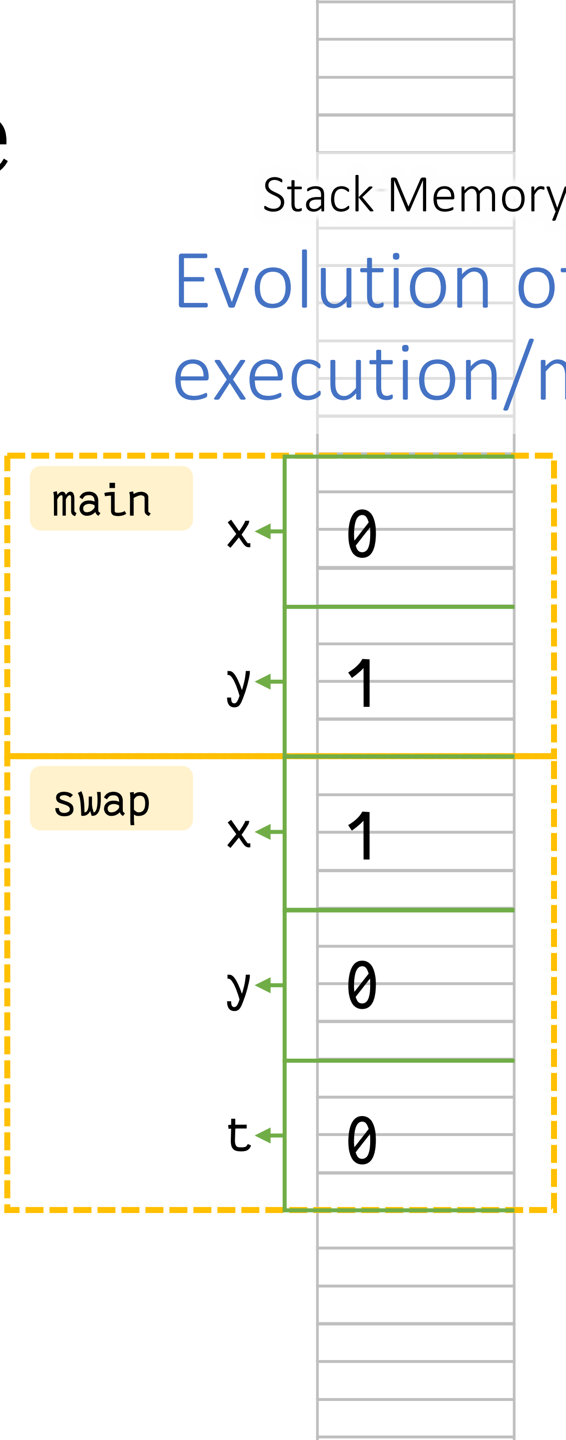
Value of the arguments are copied

C++ Memory State

Original vars not swapped

```
template <typename T>
void swap(T x, T y) {
    T t = x;
    x = y;
    y = t;
}

int main() {
    int x = 0, y = 1;
    swap(x, y);
    cout << x << y << endl;
}
```

Evolution of the execution/memory state



Pass-by-value

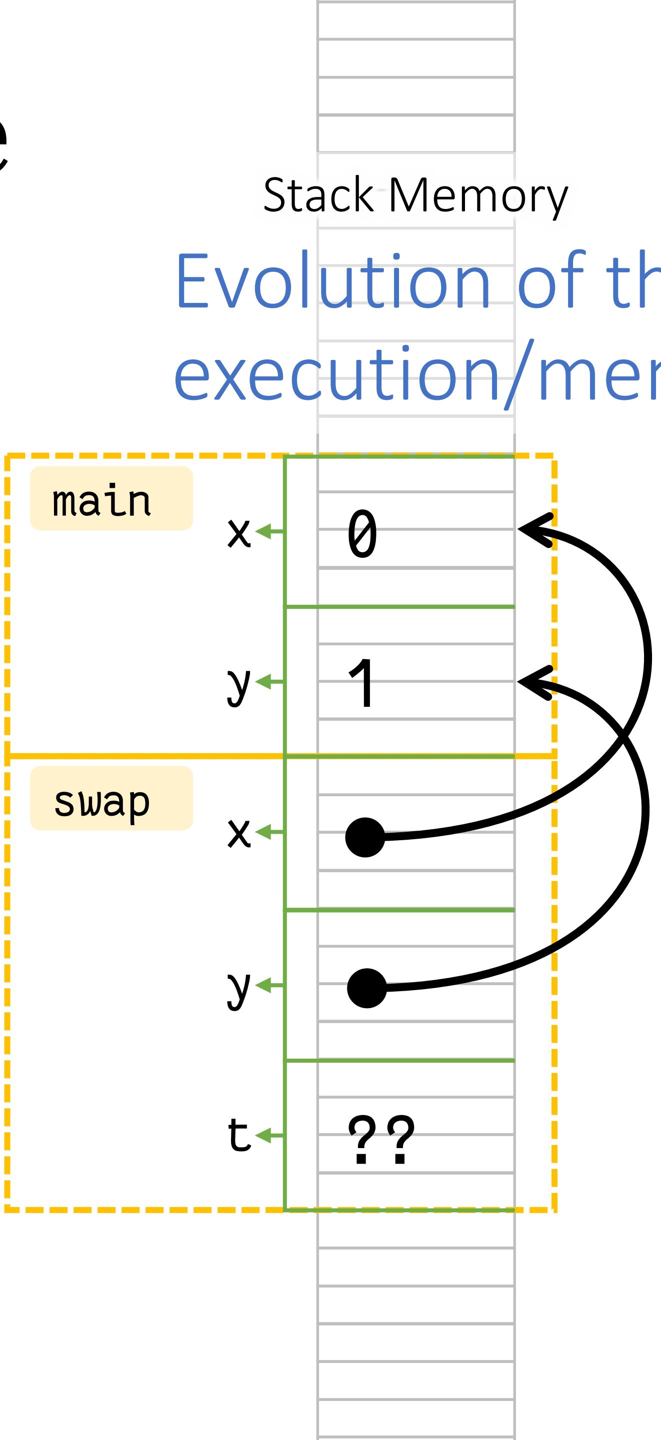
Variables are independent memory locations

C++ Memory State

Correct way to swap

```
template <typename T>
void swap(T &x, T &y) {
    T t = x;
    x = y;
    y = t;
}

int main() {
    int x = 0, y = 1;
    swap(x, y);
    cout << x << y << endl;
}
```

Evolution of the execution/memory state



Pass-by-reference

Variables refer to the arguments passed

References are not pointers

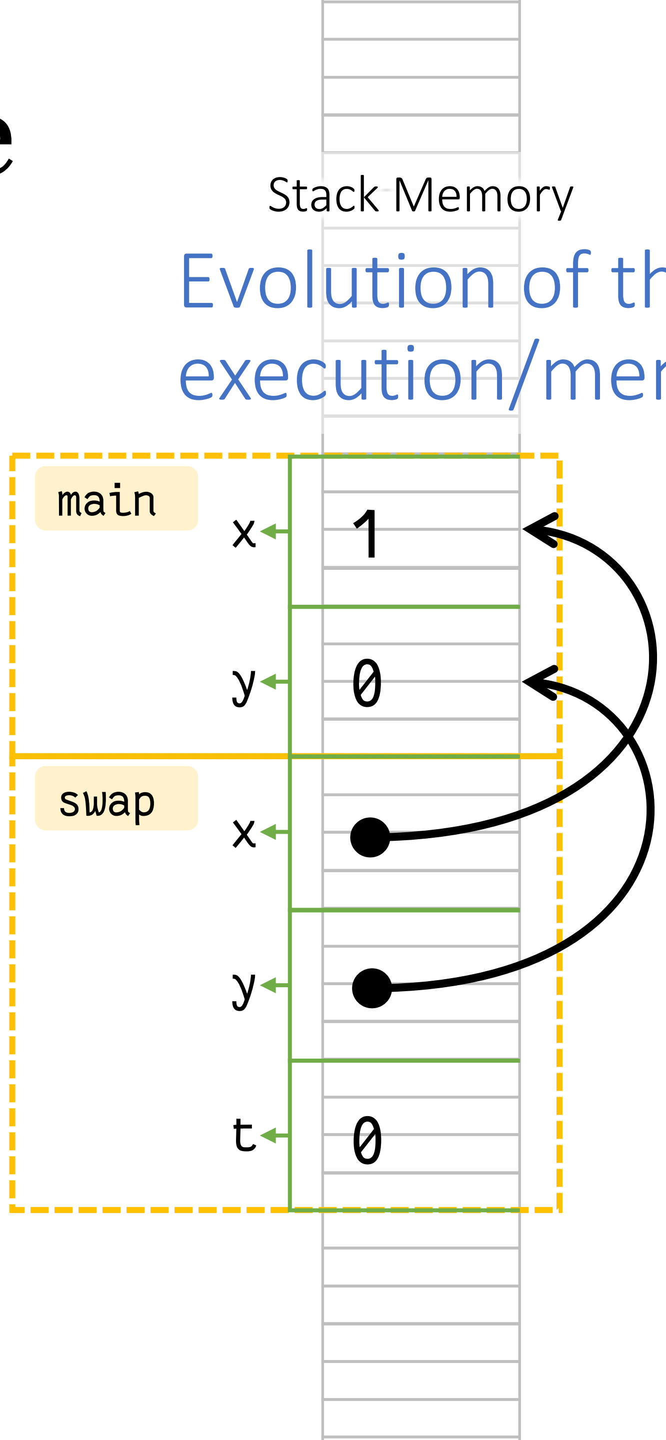
Address of x in swap is the same as address of x in main

C++ Memory State

Correct way to swap

```
template <typename T>
void swap(T &x, T &y) {
    T t = x;
    x = y;
    y = t;
}

int main() {
    int x = 0, y = 1;
    swap(x, y);
    cout << x << y << endl;
}
```

Evolution of the execution/memory state



Pass-by-reference

Variables refer to the arguments passed

References are not pointers

Address of x in swap is the same as address of x in main

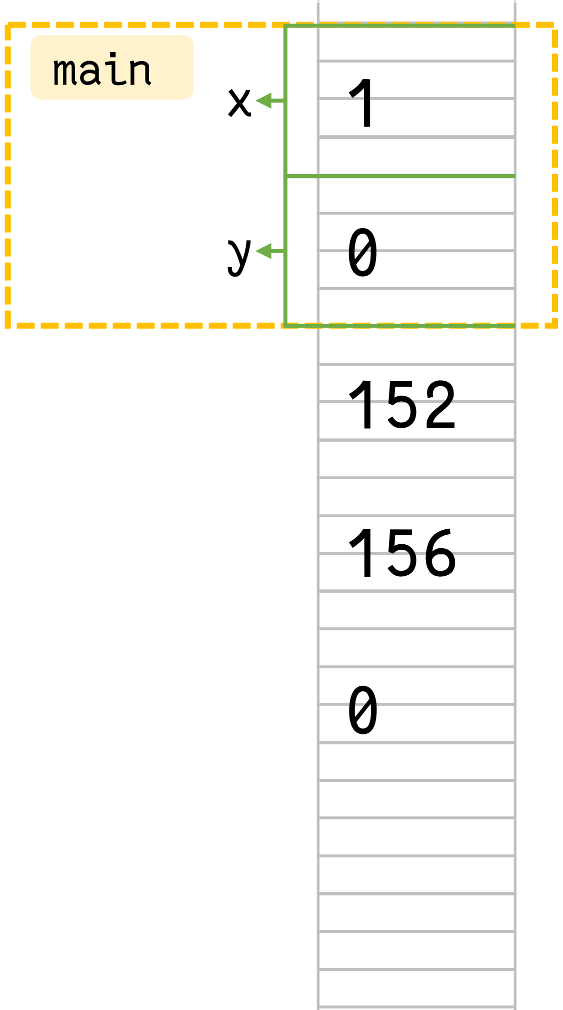
C++ Memory State

Function exits

```
template <typename T>
void swap(T &x, T &y) {
    T t = x;
    x = y;
    y = t;
}

int main() {
    int x = 0, y = 1;
    swap(x, y);
    cout << x << y << endl;
}
```

Stack Memory

Evolution of the execution/memory state



Stack Frame is deleted but the contents are not discarded/cleared.

It simply remains to be overwritten. Thus, can still be accessed by buggy/malicious code.

What about Structs?

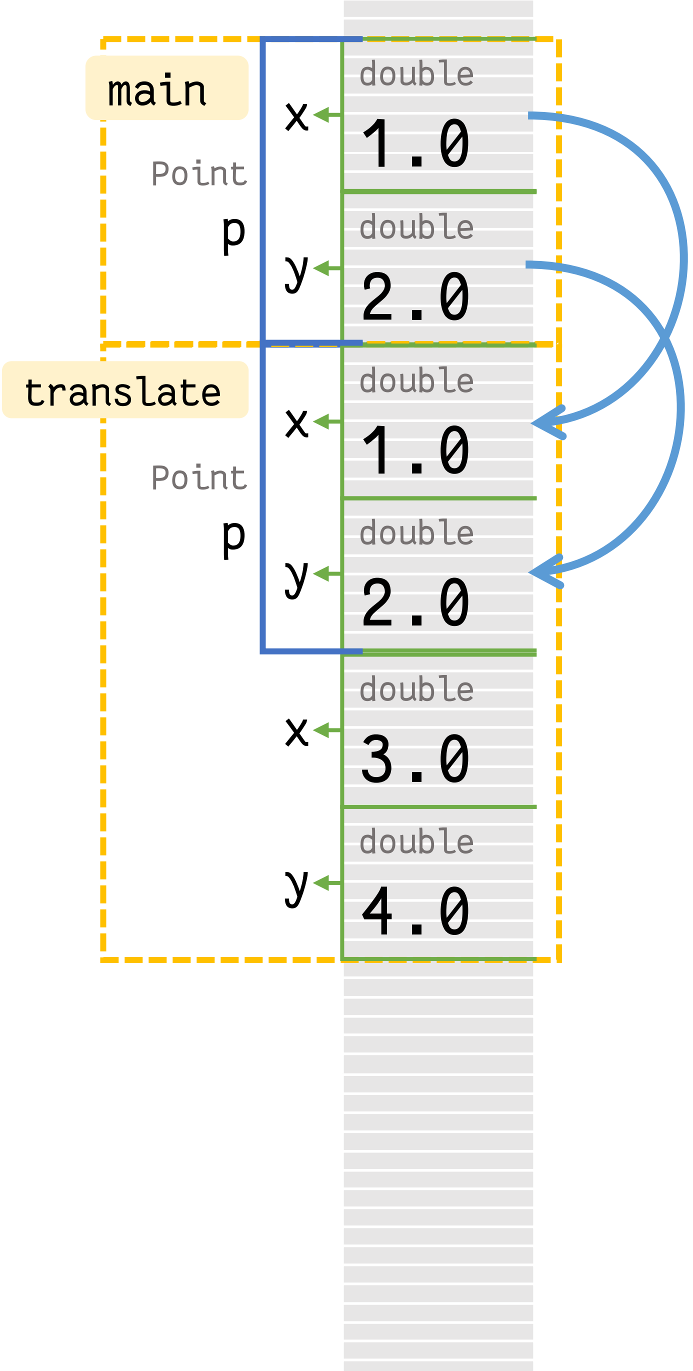
Structs are values too

```
struct Point {
    double x, y;
};

int main() {
    Point p = {1.0, 2.0};  // break abstraction
    Point q = p;
    p.x = 5.0;
    cout << q.x << "," << q.y;
}
```

Passing Structs

With pass-by-value, contents are copied

```
void translate(Point p, double x, double y) {
    p.x += x;    // break abstraction
    p.y += y;    // for simplification
}

int main() {
    Point p = {1.0, 2.0};
    translate(p, 3.0, 4.0);
}
```



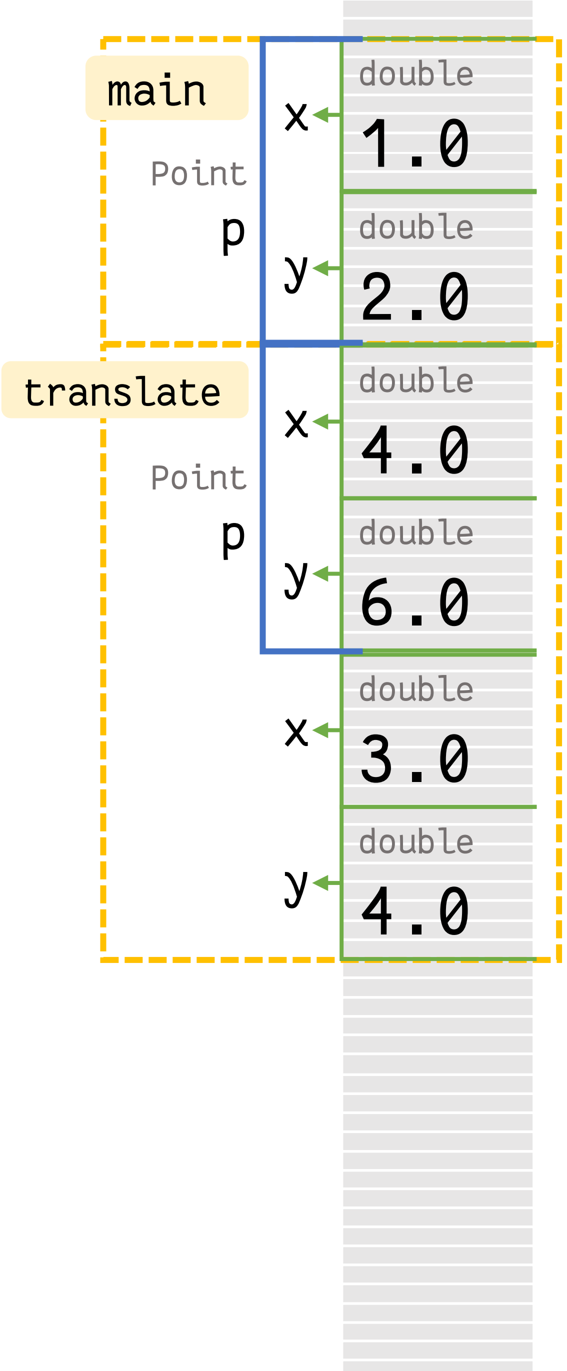
https://goo.gl/no69wh

Passing Structs

With pass-by-value, contents are copied

```
void translate(Point p, double x, double y) {
    p.x += x;    // break abstraction
    p.y += y;    // for simplification
}

int main() {
    Point p = {1.0, 2.0};
    translate(p, 3.0, 4.0);
}
```

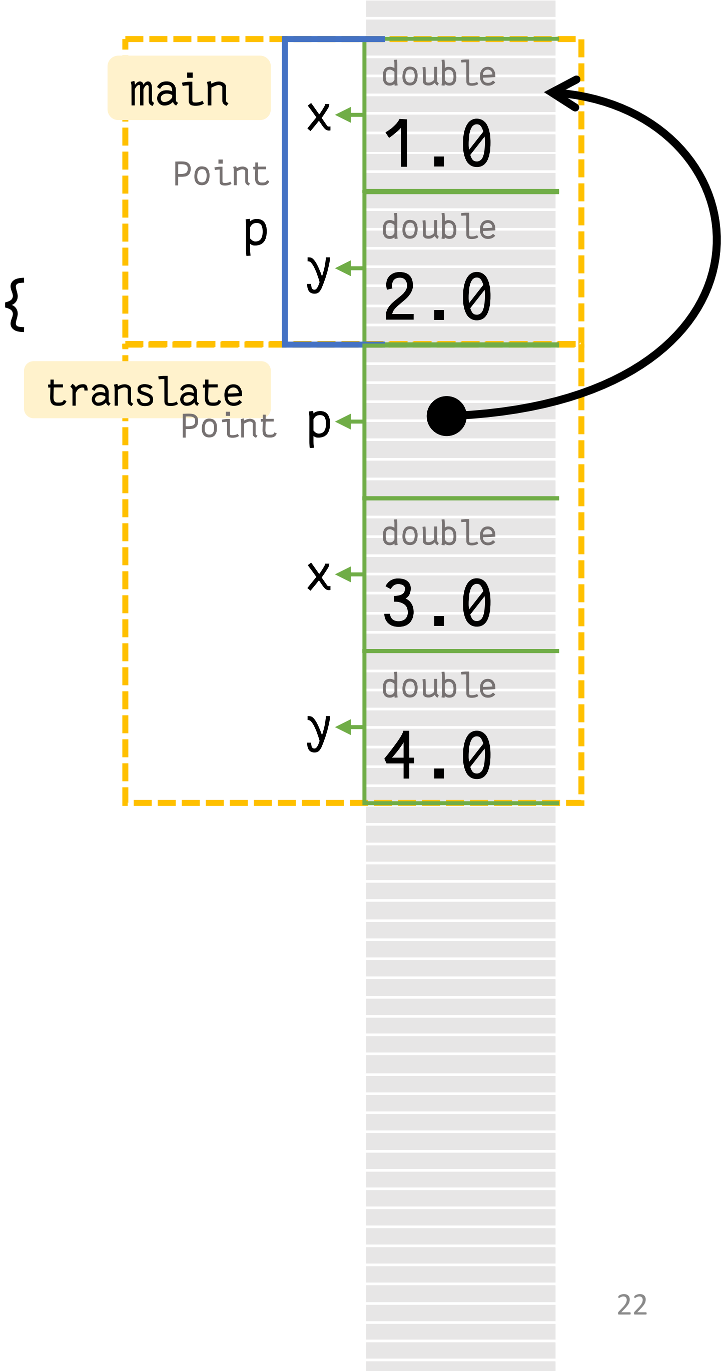
No modification is done



Passing Structs

That is why we use pass-by-reference

```
void translate(Point &p, double x, double y) {
    p.x += x;   // break abstraction
    p.y += y;   // for simplification
}

int main() {
    Point p = {1.0, 2.0};
    translate(p, 3.0, 4.0);
}
```



Passing Structs

What about constant arguments?

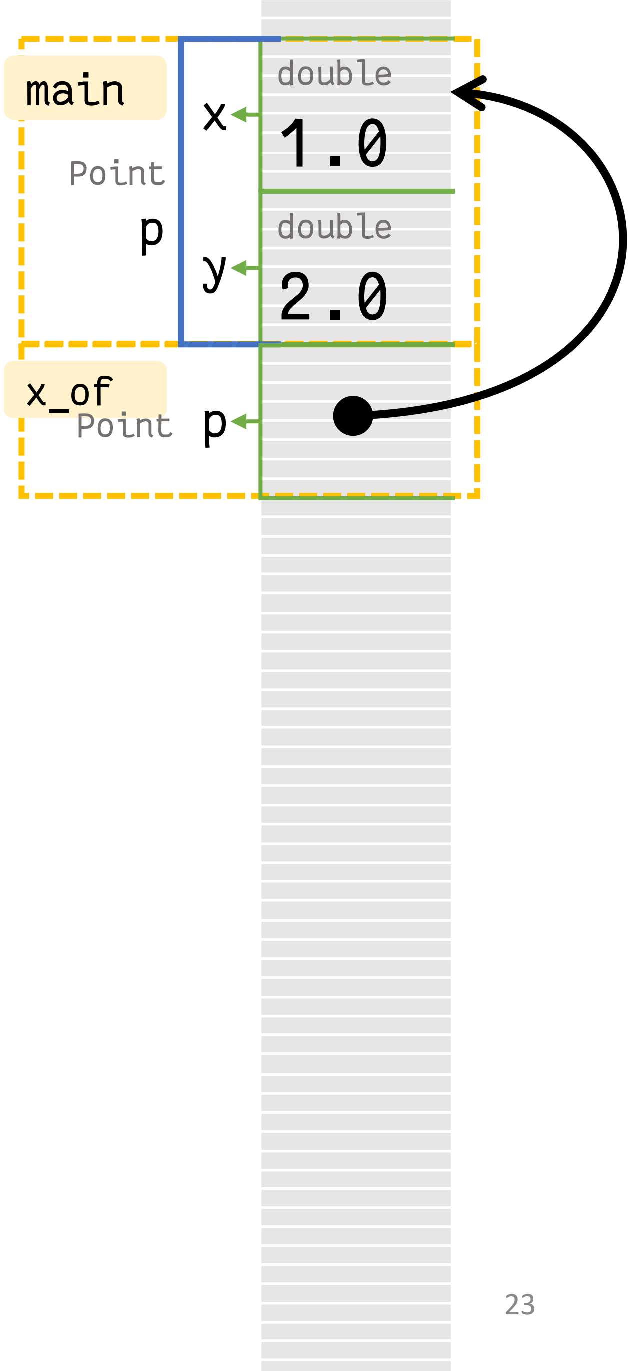
```
void x_of(Point &p) {
    return p.x;
}
```

– Why use reference when there is no modification?

No copying is needed

– Save time and space if contents are large, O(1)

– const indicate that argument will not be modified



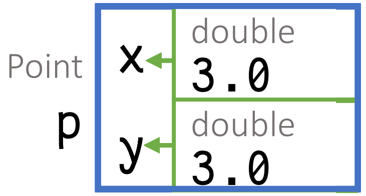
What about Vectors?

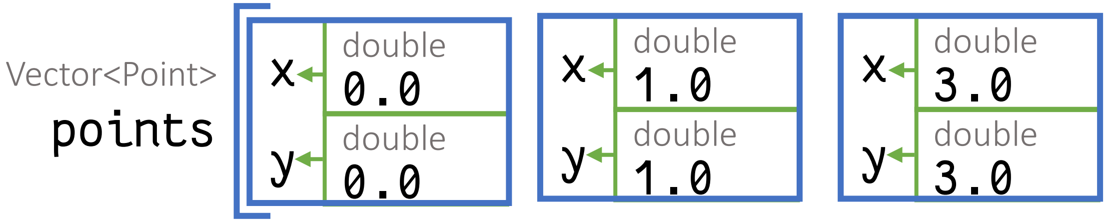
Yes! Vectors are values too

```
vector<int> v = {1, 2, 3, 4};
vector<int> w = v;
v[0] = 42;

for (int i : v)
    cout << i << " ";
cout << endl;

for (int i : w)
    cout << i << " ";
cout << endl;
```

Recall Vector of Points

```
Point p;
vector<Point> points;

make_point(p, 0, 0);
for (int i=0; i < 5; i++) {
    translate(p, i, i);
    vector.push_back(p);
}
```





Vector push_back creates a copy

– points contains different Points

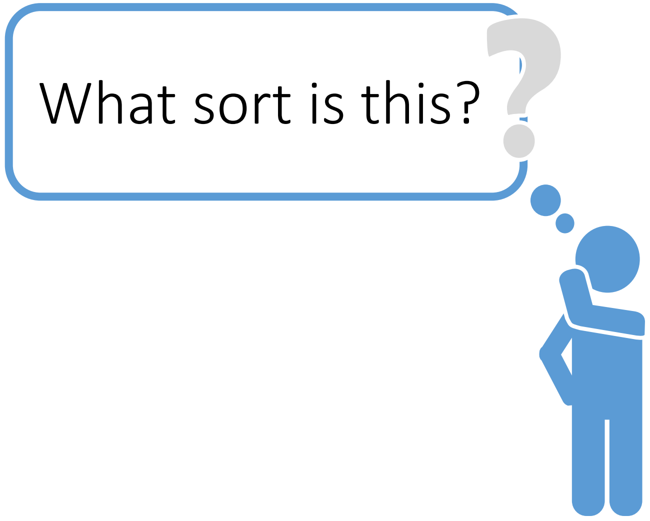
Passing into Functions

- Same as structs, contents of vectors are copied
- Which is why using references can save time
  - $O(1)$ instead of $O(n)$, where n is size of container

Passing into Functions

Same as structs, contents of vectors are copied

```
void sort(vector<int> v) {
    bool done = false;
    while (!done) {
        done = true;
        for (int i = 0; i < v.size()-1; i++)
            if (v[i] > v[i+1]) {
                swap(v[i], v[i+1]);
                done = false;
            }
    }
}
```



Passing into Functions

Same as structs, contents of vectors are copied

Which is why using references can save time

- $O(1)$ instead of $O(n)$, where n is size of container

In fact,

- same goes for map, set, ...
- and any other struct or objects

Except....

- for arrays

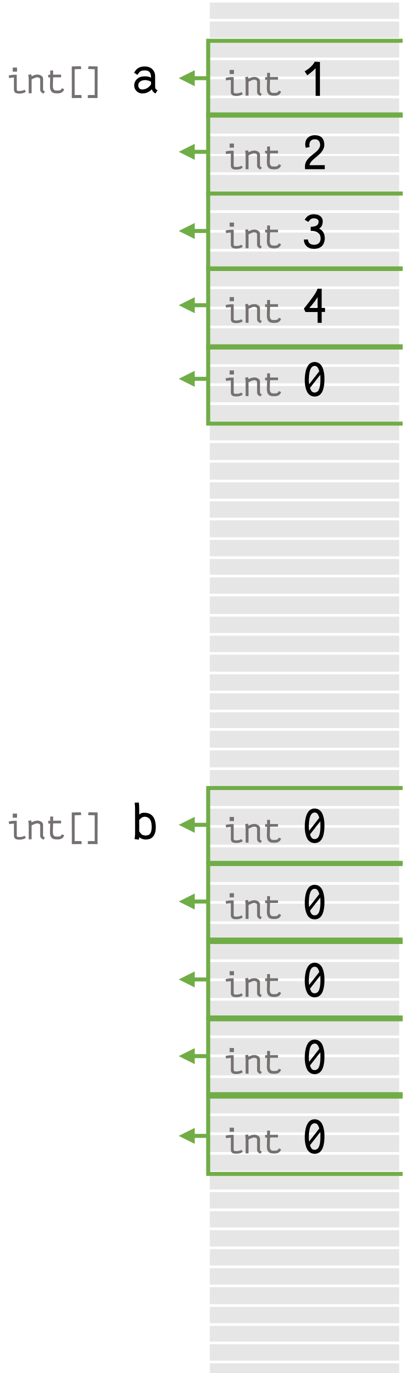
Arrays in C/C++

Arrays can NEVER, ever be passed by value

– because they are always addresses (i.e. pointers)

```
int a[5] = {1, 2, 3, 4, 5};
int b[5] = {0};

b = a;
```

Compile Error. Invalid array assignment

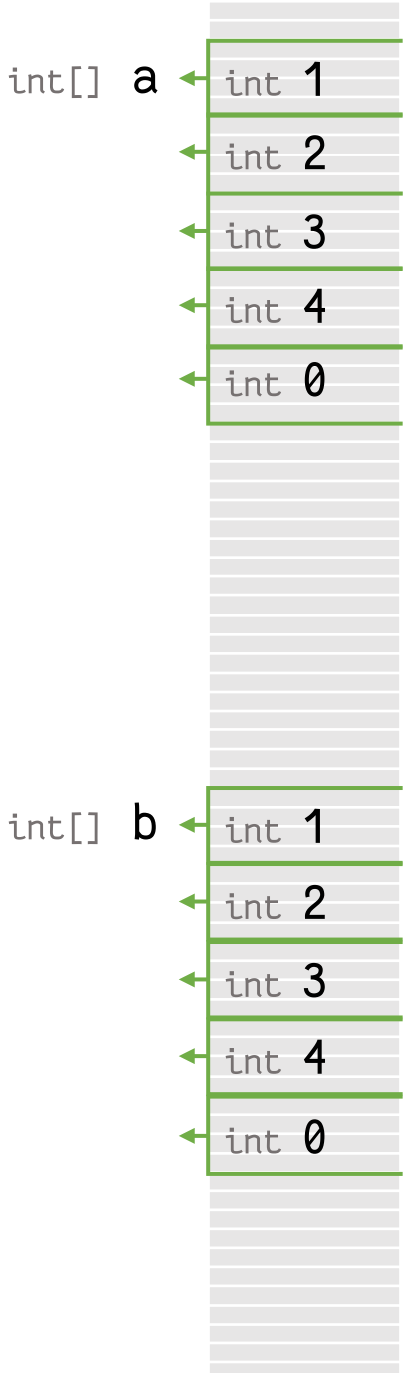


Arrays in C/C++

Arrays can NEVER, ever be passed by value

- because they are always addresses (i.e. pointers)
- need to manually copy the array

```
int a[5] = {1, 2, 3, 4};
int b[5];

for (int i = 0; i < 5; i++)
    b[i] = a[i];

// or use memcpy
memcpy(b, a, sizeof(a));
```

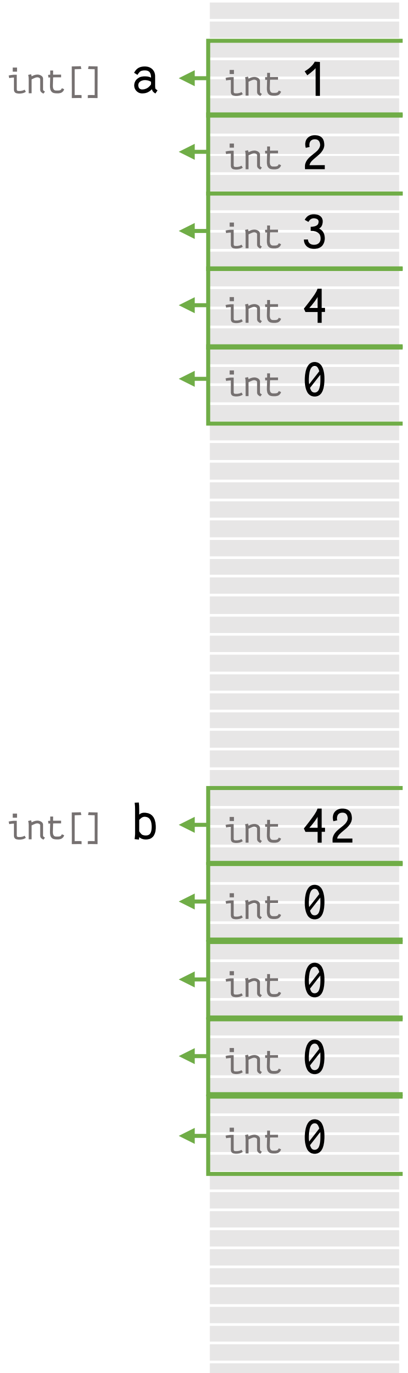


Arrays in C/C++

Arrays have no limits

– No bound on accessing array elements
– Compiler will let you access where ever you want

```
int a[5] = {1, 2, 3, 4, 5};
int b[5] = {0};

cout << a << endl;
cout << b << endl;

a[8] = 42;
cout << b[0] << endl;
```



Passing Arrays into Functions

Cannot be passed by value

no difference from `int *v`

```
void sort(int v[], int size) {
    for (bool flag = false; flag = !flag)
        for (int i = 0; i < size()-1; i++)
            if (v[i] > v[i+1]) {
                swap(v[i], v[i+1]);
                flag = false;
            }
}
```

This is the same sort as previously, only much more compact code

Re-examining Constructors

Recall how we wrote constructors

```
void make_point(Point &p, double x, double y);
```

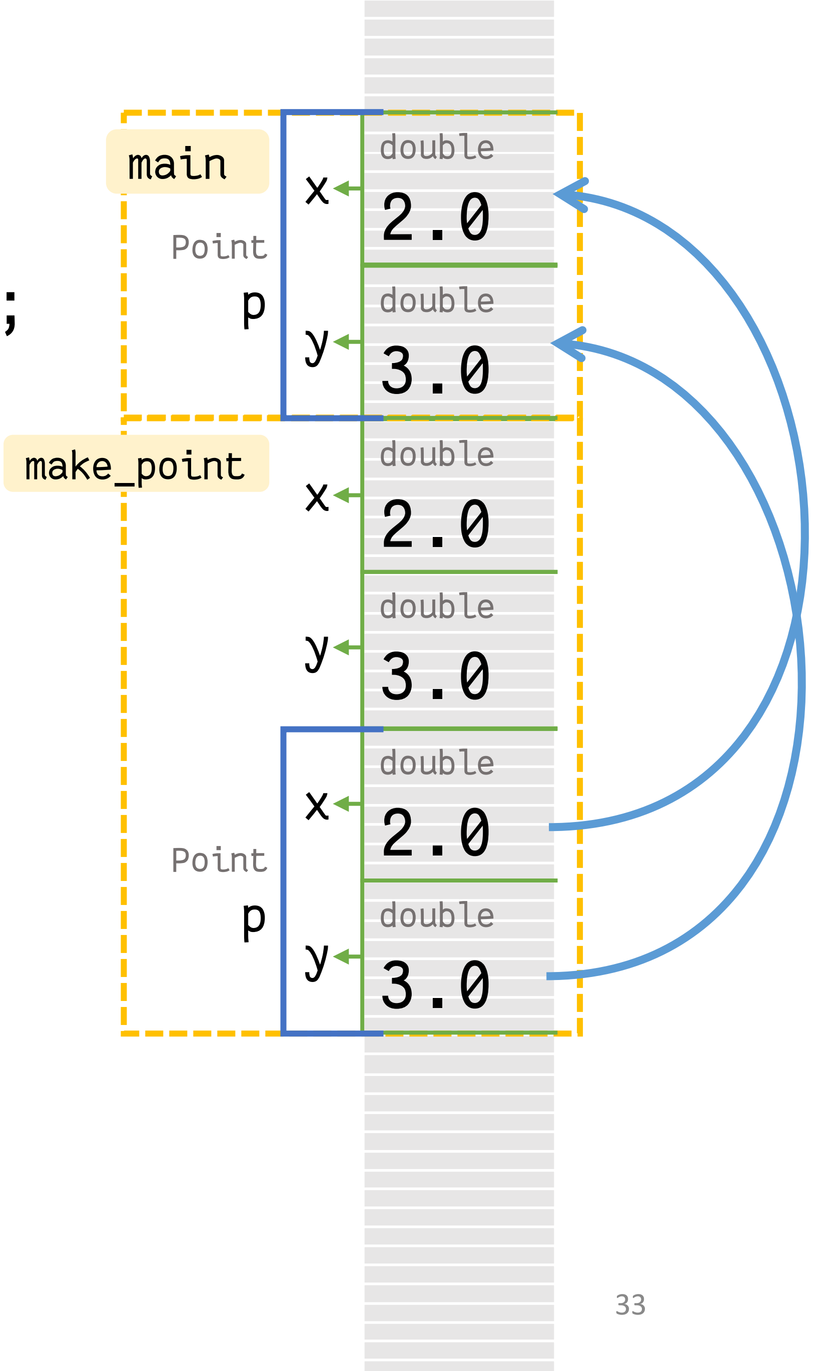
– Quite unintuitive as we would have already “created” a Point to pass into the function

```
Point make_point(double x, double y) {
    Point p = {x, y};
    return p;
}
```

– This constructor returns a new Point

– Returned value is copied into calling Point

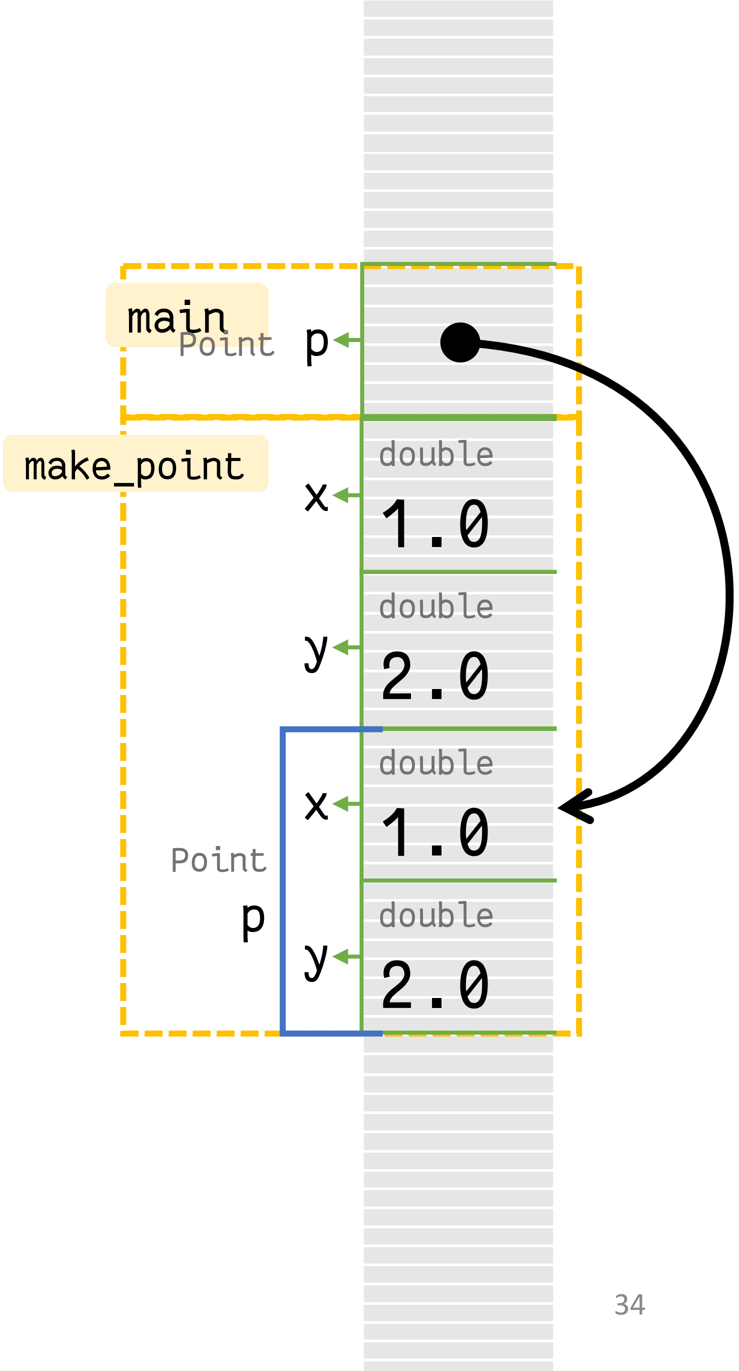
```
Point p = make_point(2.0, 3.0);
```



Returning References?

What if constructors return by reference?

```
Point & make_point(double x, double y) {
    Point p = {x, y};
    return p;
}

int main() {
    Point &p = make_point(1.0, 2.0);
}
```

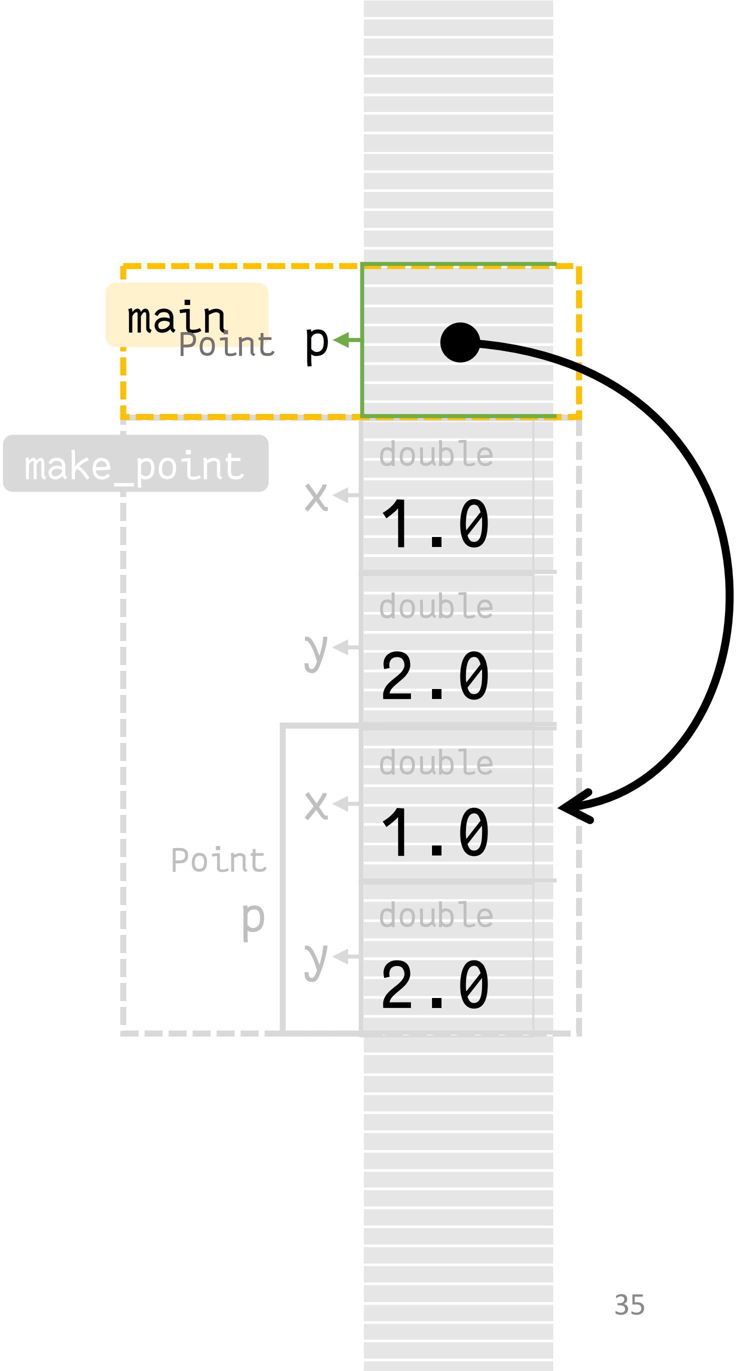


Returning References?

What if constructors return by reference?

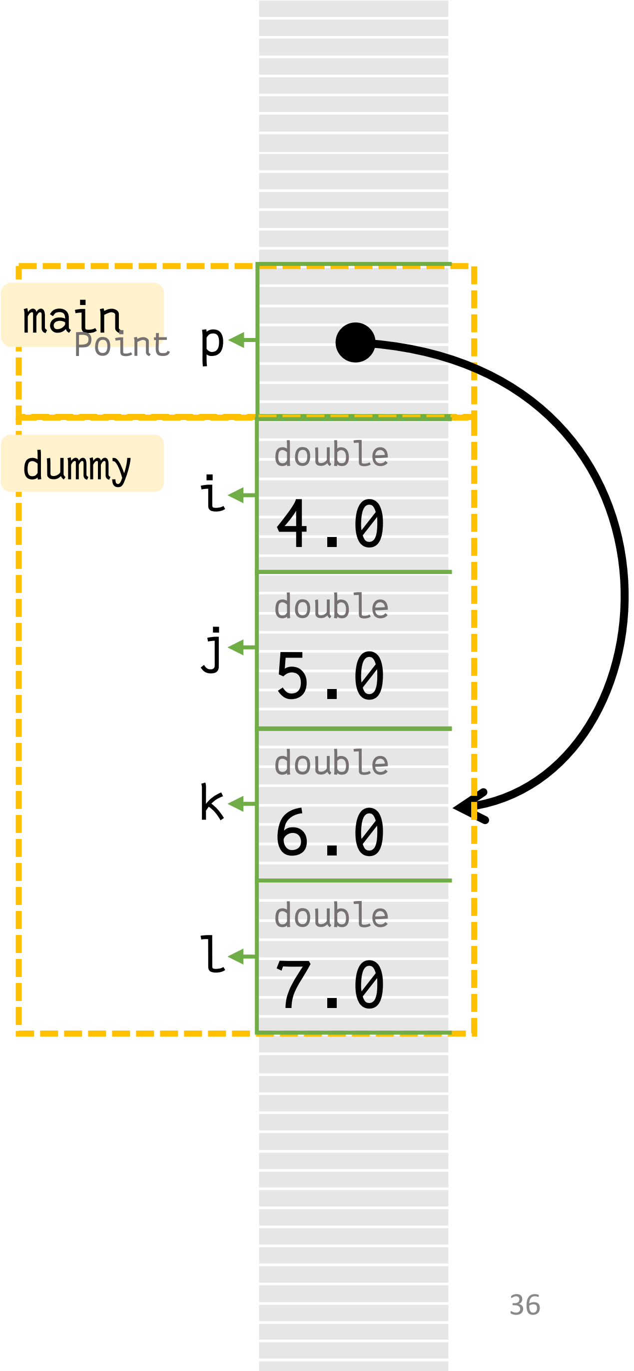
```
Point & make_point(double x, double y) {
    Point p = {x, y};
    return p;
}

int main() {
    Point &p = make_point(1.0, 2.0);
}
```

What happens when the function exits?



Returning References?

```
void dummy() {
    double i=4, j=5, k=6, l=7;
}

int main() {
    Point &p = make_point(1.0, 2.0);
    dummy();
    cout << p.x << p.y << endl;
}
```

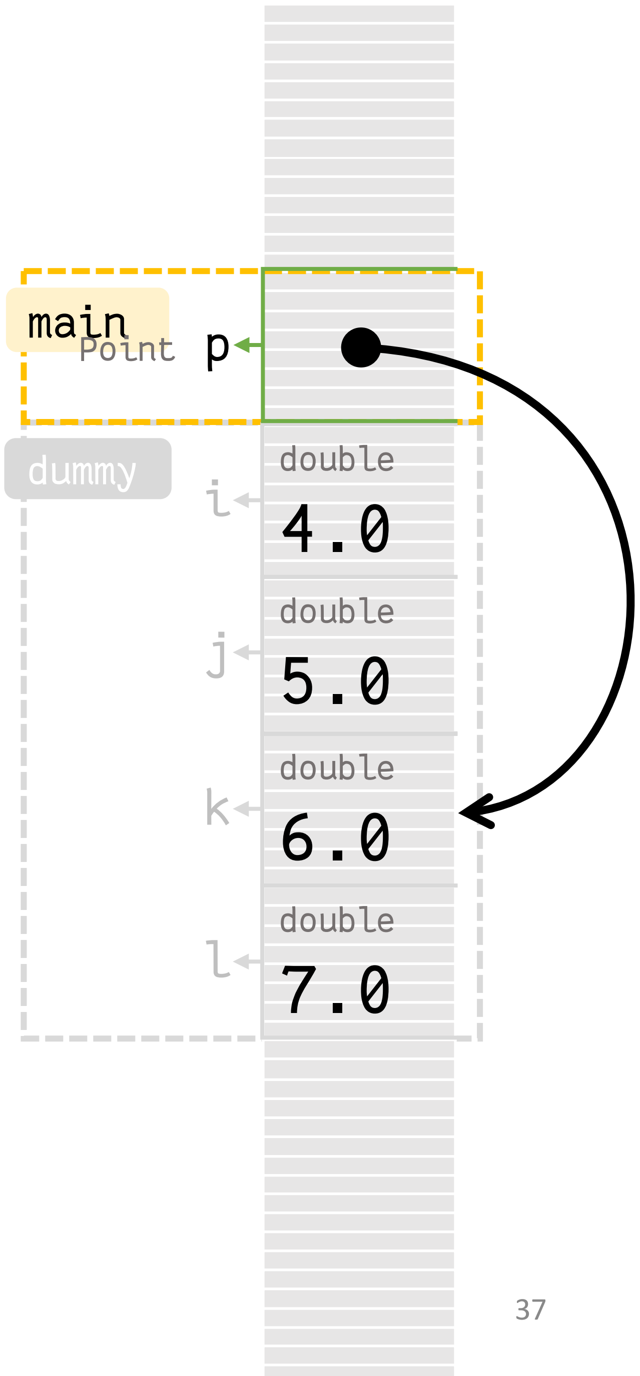


What happens when the function exits?

– And another function is called?

Returning References?

```
void dummy() {
    double i=4, j=5, k=6, l=7;
}

int main() {
    Point &p = make_point(1.0, 2.0);
    dummy();
    cout << p.x << p.y << endl;
}
```



Memory values get overwritten

– What are the values of p.x and p.y now?

Why is all this important?

Because C++

- gives complete control to programmer
- does not manage memory
- reduce housekeeping overhead

Thus, the programmer needs to

- be aware of potential pitfalls
- be mindful when debugging
- manage memory on the heap

Recursive Containers

What if a struct can contain itself?

Recall: Queue ADT

Two implementations

- Using vector
- Using two stacks

Time complexity is at least linear

- because of dequeue

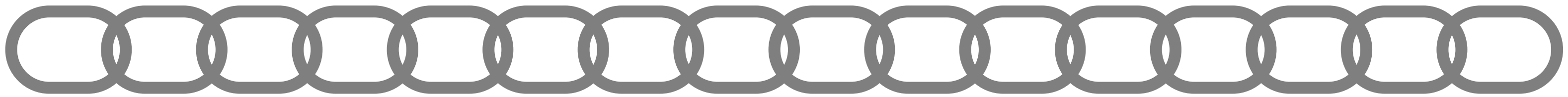
Vectors (and arrays) have similar issue

- Random accessing is very fast
- Removal from anywhere but the back is slow

Linked Structure

Imagine some linked structure

- like a chain, or polymer, or DNA, etc.



What do you notice about these kinds of structure?

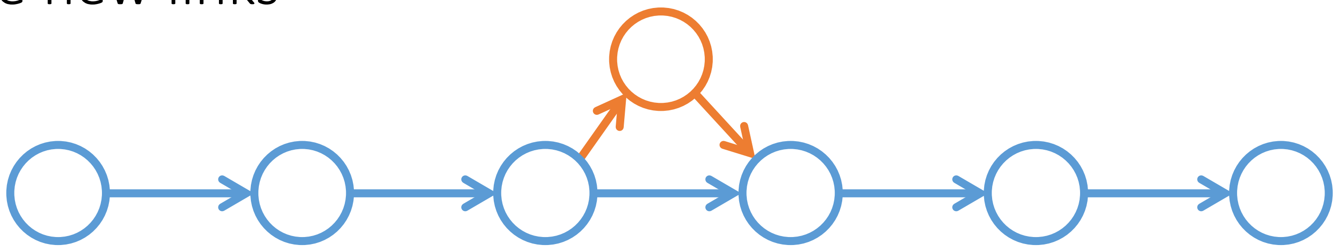
- It is repetitive, the same unit repeated
- Each unit links to one other unit

What's so good about these structures?

Linked Structures

How to add new link/node?

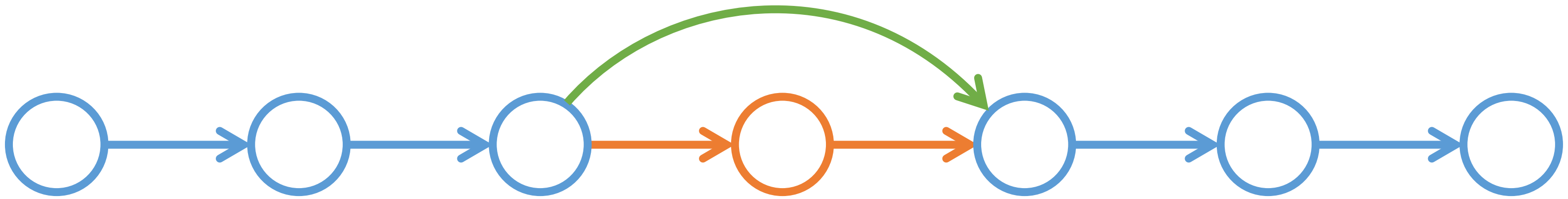
- Break existing link
- then recreate new links



Linked Structures

How to add new link/node?

– Break existing link
– then recreate new links

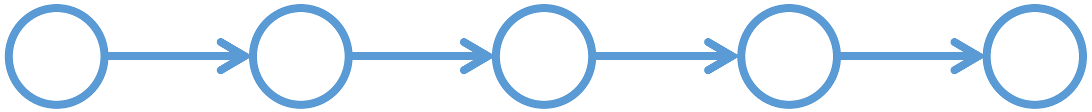


How to remove existing link/node?

– Re-route existing link
– Remove node

What is the complexity? $O(1)$

Linked List ADT



Defining a list node

- node should be a container, some item
- node should link to next node

```
template <typename T>
struct ListNode {
    T item;
    ListNode next;
};
```

Compile Error. Field has incomplete type

What is wrong?

Defining Self-referencing Structures

Memory for struct must be allocated

- thus compiler needs to know size of struct at compile time
- same reason why forward declarations are needed

```
void A(int i);

void B(int i) {
    A(i+i);
}
```

How much space to allocate?

```
struct LinkNode {
    T item;
    LinkNode next;
};
```

size of T is known

?? bytes

- LinkNode contains a LinkNode, which contains a LinkNode…
- ad infinitum!

Defining Self-referencing Structures

Use a pointer

```
template <typename T>
struct LinkNode {
    T item;
    LinkNode *next;
};
```

4 bytes if T is int

8 bytes for pointer

The link to next node

- is really a link/reference/pointer

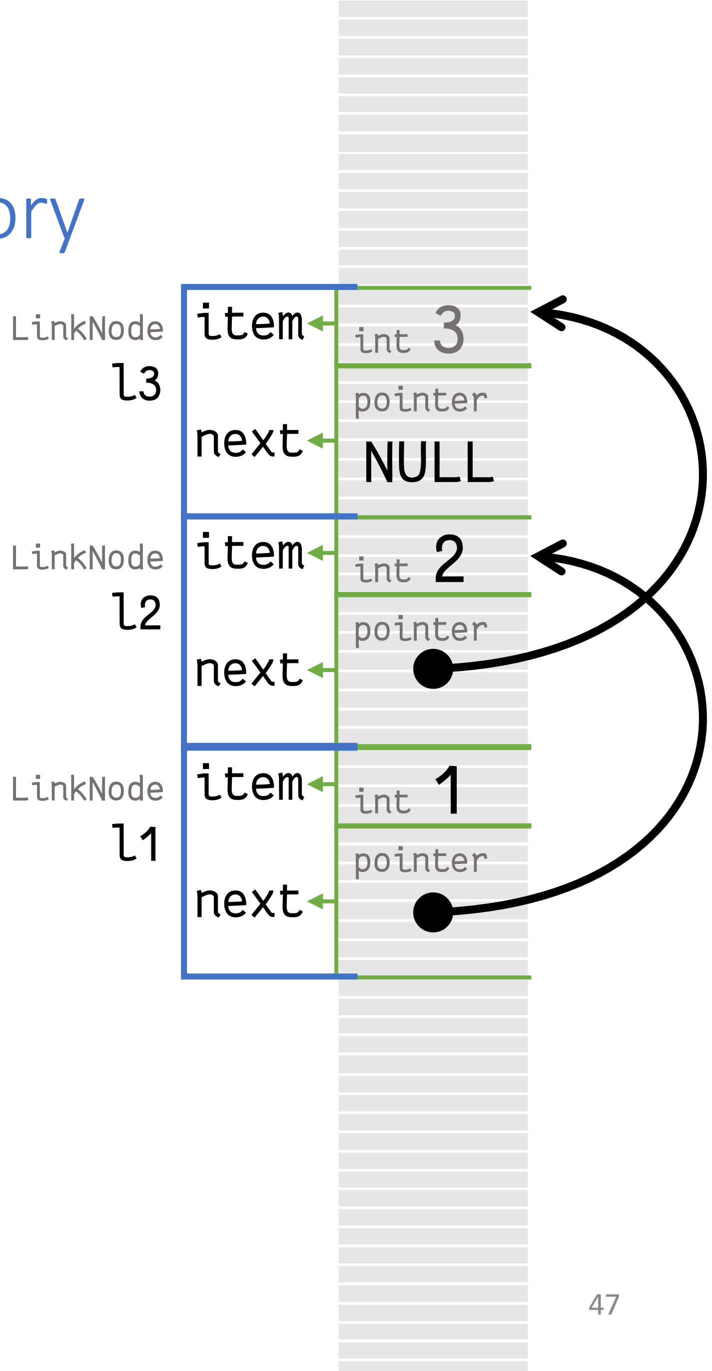
Static Allocation

Declaring every node

```
int main() {
  LinkNode<int> l3 = {3, NULL},
                l2 = {2, &l3},
                l1 = {1, &l2};


}
```

- If every node has to be declared
- Might as well use array
- Very complex to manage add and remove

Stack Memory



https://goo.gl/oeJ5jq

Allocating Nodes

Inserting new nodes

```
void insert_node(LinkNode<T> &new_node, LinkNode<T> &after) {
    new_node.next = after.next;
    after.next = &new_node;
}
```

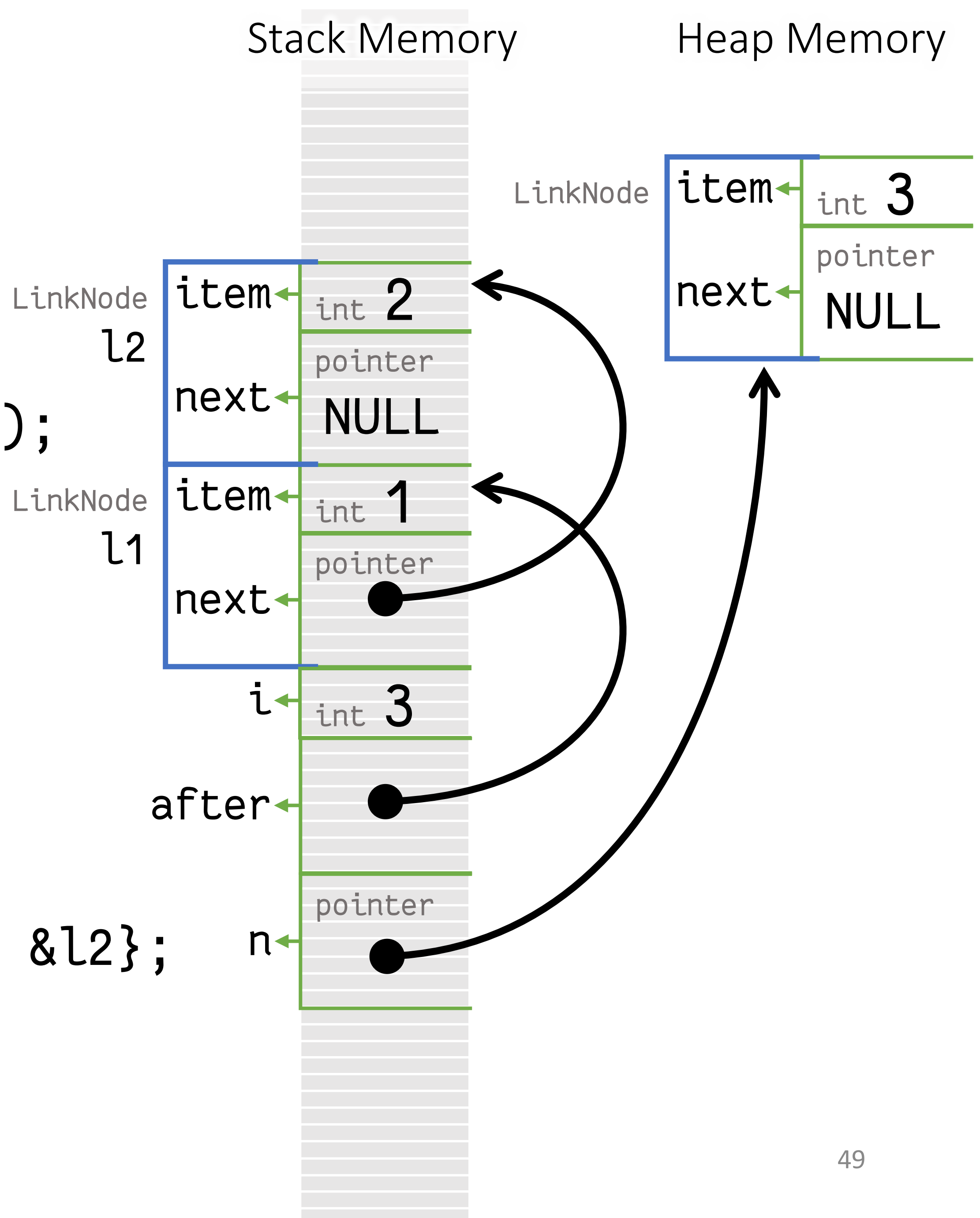
– caller has to create a new node
– memory allocated on stack does not persist across functions

Solution: allocate on the heap

Dynamic Allocation

Allocate on heap using `new`

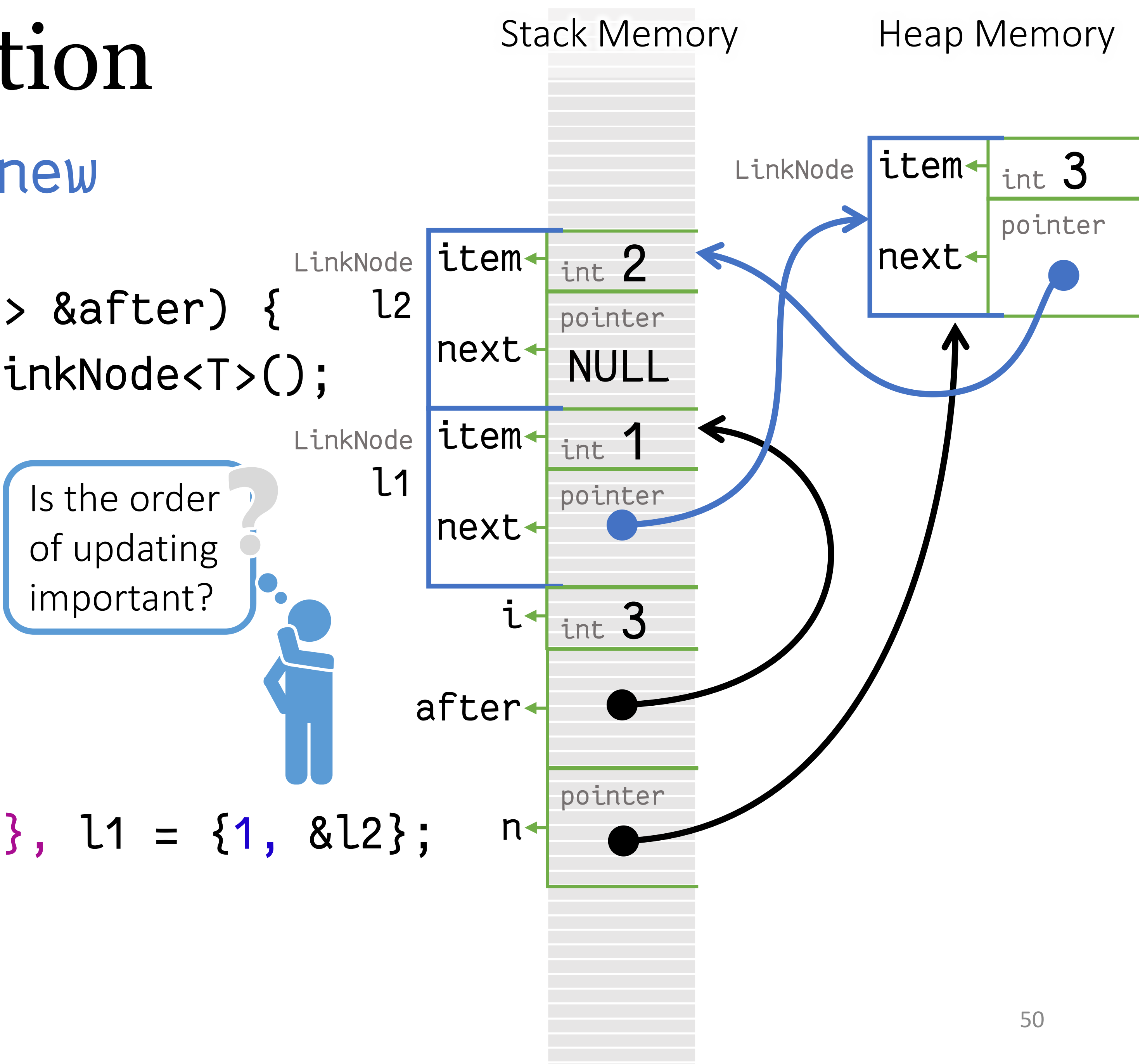
```
void add_node(T item,
              LinkNode<T> &after) {
  LinkNode<T> *n = new LinkNode<T>();
  n->item = item
  n->next = after.next;
  after.next = n;
}

int main() {
  LinkNode l2 = {2, NULL}, l1 = {1, &l2};
  add_node(3, l1);
}
```



Dynamic Allocation

Allocate on heap using new

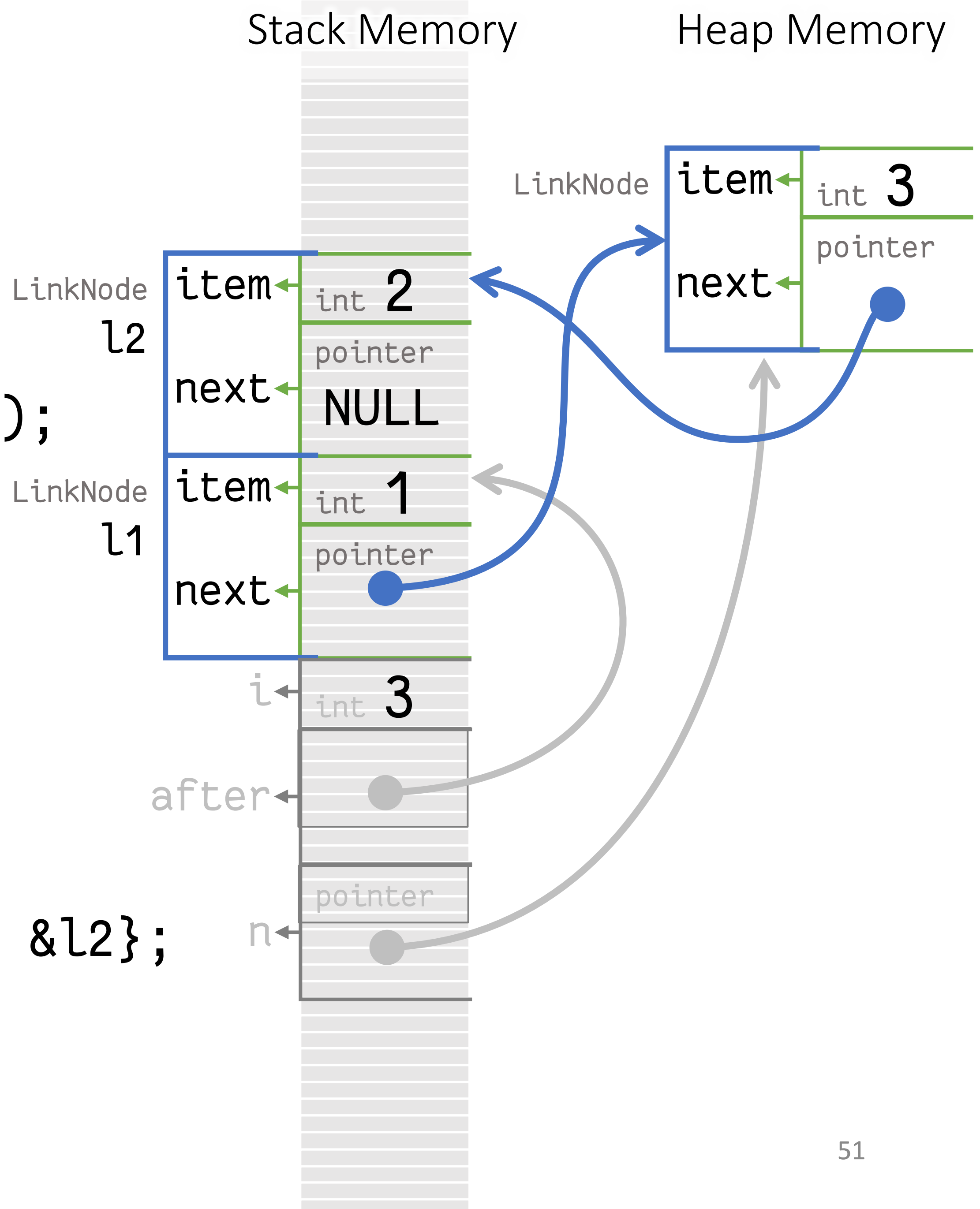
```
void add_node(T item,
              LinkNode<T> &after) {
  LinkNode<T> *n = new LinkNode<T>();
  n->item = item
  n->next = after.next;
  after.next = n;
}

int main() {
  LinkNode l2 = {2, NULL}, l1 = {1, &l2};
  add_node(3, l1);
}
```



Dynamic Allocation

Allocate on heap using **new**

```
void add_node(T item,
              LinkNode<T> &after) {
  LinkNode<T> *n = new LinkNode<T>();
  n->item = item
  n->next = after.next;
  after.next = n;
}

int main() {
  LinkNode l2 = {2, NULL}, l1 = {1, &l2};
  add_node(3, l1);
}
```



Dynamic Allocation

Memory allocated on heap

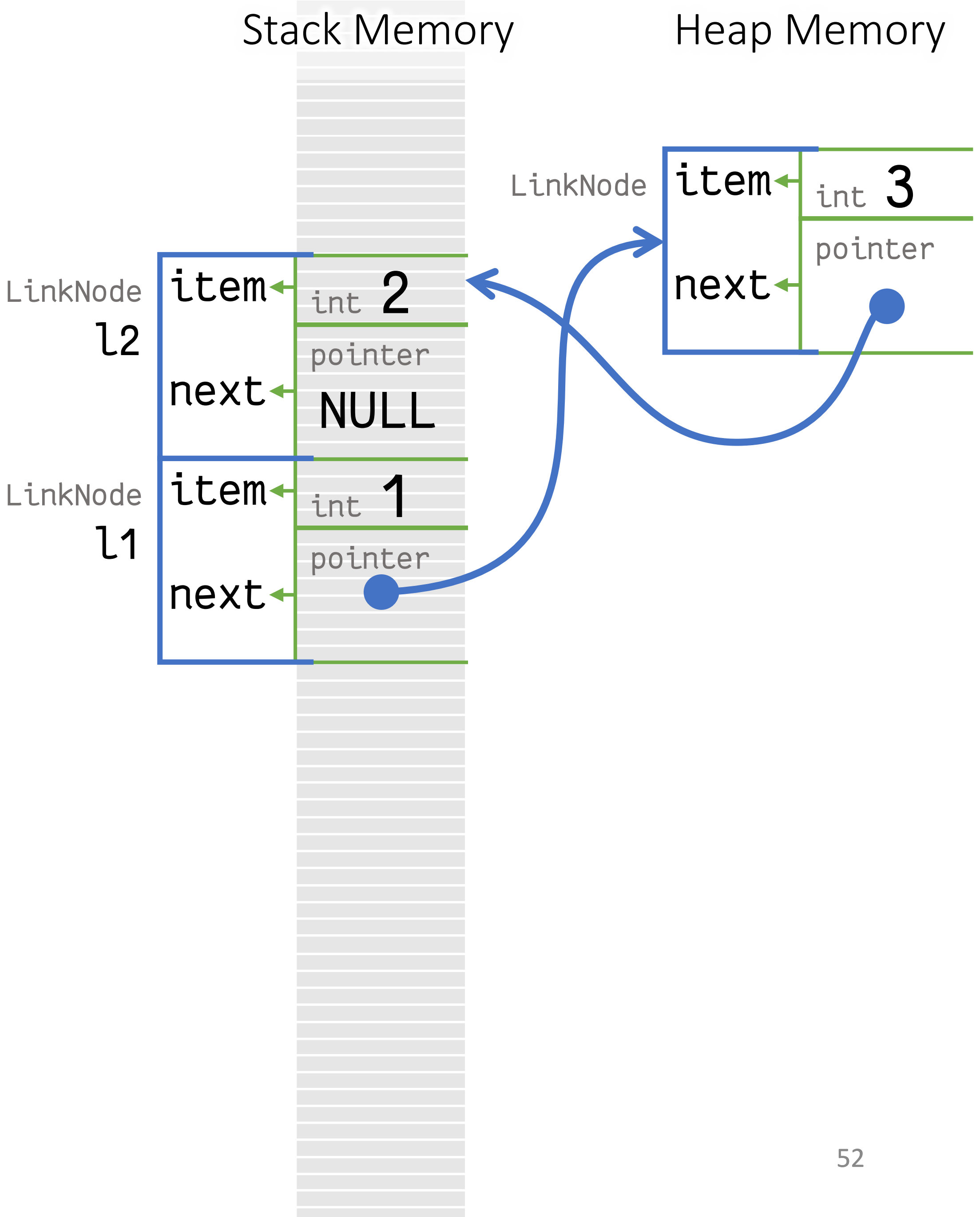
- Persists after stack frame is removed
- must be explicitly removed using delete

```
delete l1.next;
```

- As usual, contents are not cleared

One caveat

- You must know what to free
- Losing the reference means memory is never freed
- leading to memory leak



Dynamic Memory Allocation

– aka Runtime Allocation

To allocate

– Use new

To unallocate

– Use delete

Required when size of data (e.g. number of elements) is not known at compile time

– More flexible use of space rather than over reserve

Searching in Linked List

Given the head node, look for item

- How to iterate down the chain?
- What do you notice about the chain?

- Cutting off part of the chain, it’s still a chain

Strategy

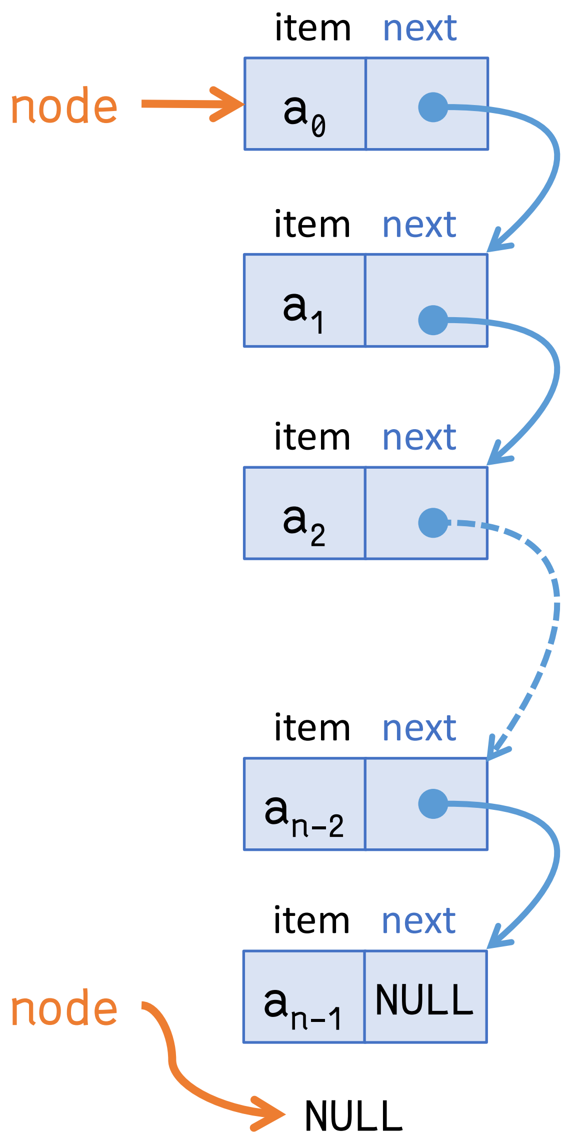
- Check if head has the item
- If not, cut it off and repeat, until no more chain

Searching with Recursion

Idea problem to use recursion

```
template <typename T>
bool find(LinkNode<T> *node, T item) {
    if (node == NULL)
        return false;
    if (node->item == item)
        return true;
    else
        return find(node->next, item);
}
```

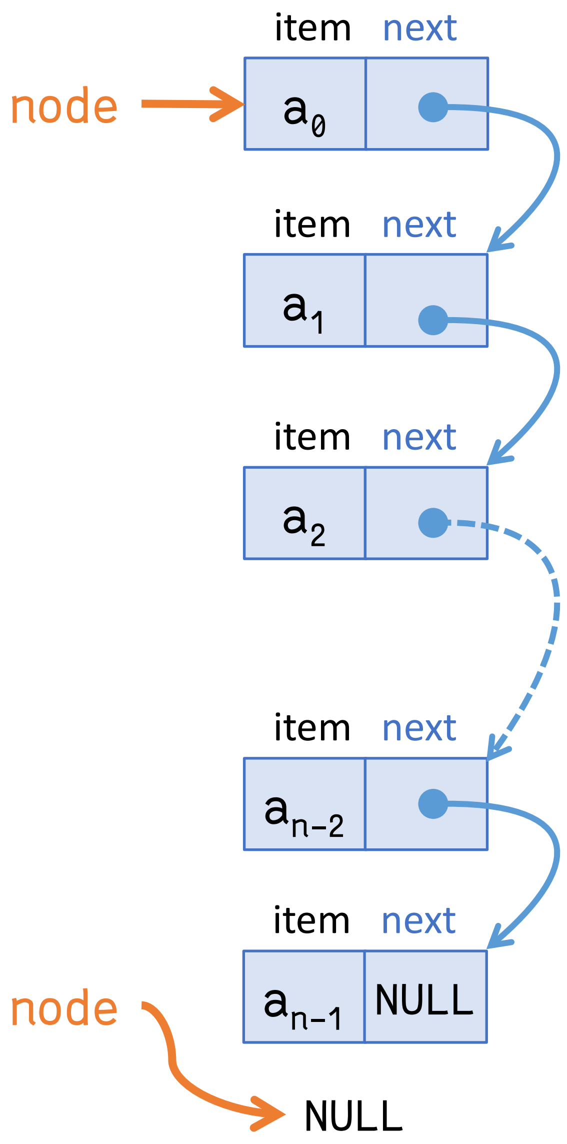
– Can also write using reference



Searching with Iteration

This problem is easy enough to use iteration

```
template <typename T>
bool find(LinkNode<T> *node, T item) {
    while (node != NULL) {
        if (node->item == item)
            return true;
        else
            node = node->next;
    }
    return false;
}
```



Queue ADT: Using Linked List

Implementing a queue with an linked list as the internal data

Recall: Homemade Queue ADT

1. Using a vector
2. Using two stacks
3. Using a linked list

	Vector	2 stacks	Linked List
Enqueue	$O(1)$	$O(1)$	?
Dequeue	$O(n)$	$O(1)\sim O(n)$	?

Queue ADT using Linked List

Can we just use a ListNode as head of a Queue

- Absolutely
- It is after all a sequence structure, which is what a queue is

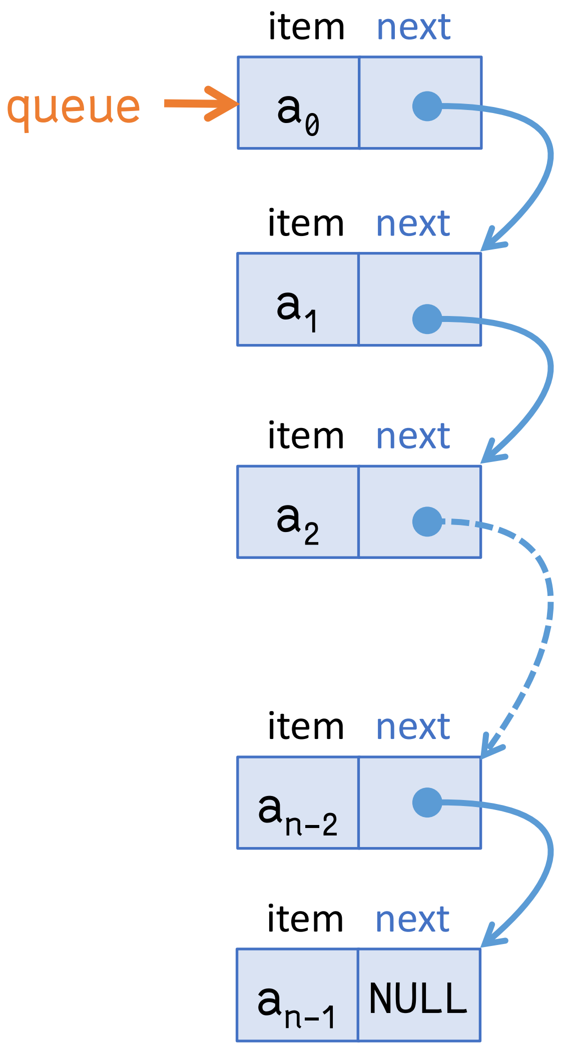
Dequeue is now O(1), excellent!

```
queue = queue->next;  // don't forget to delete
```

What about Enqueue?

- Must transverse until the end of the queue to the last node
- O(n), where n is the length of queue

Same for obtaining size of queue



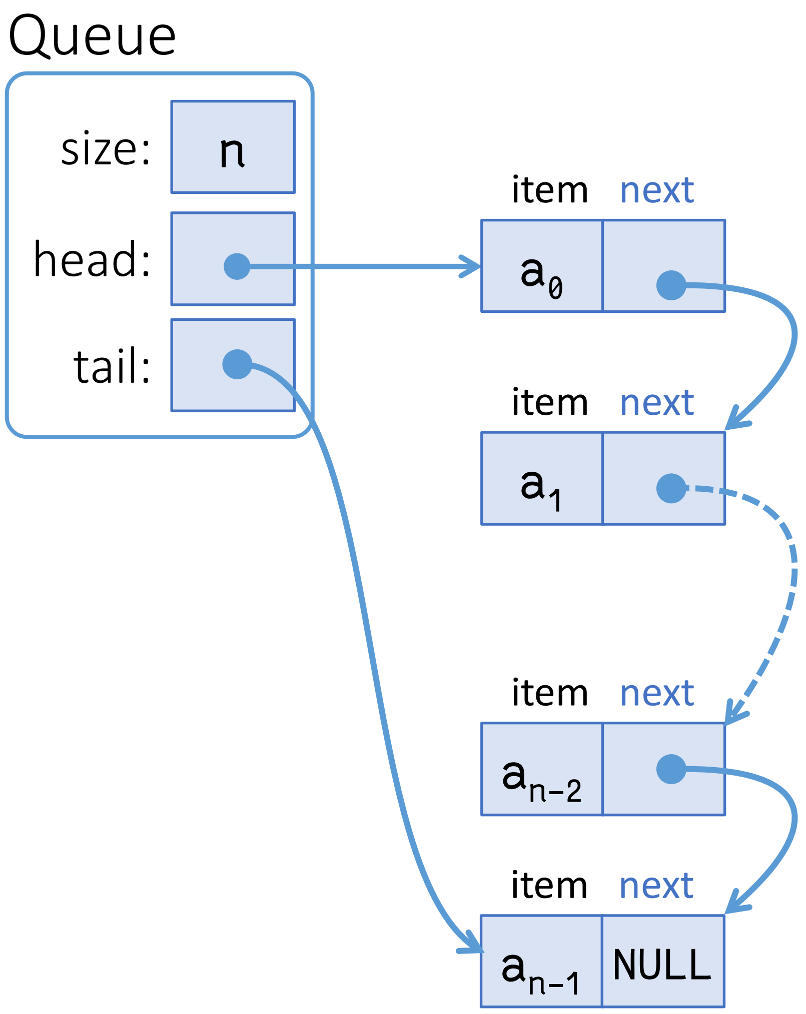
Wrapping the Linked List

Thankfully ADTs can hide complexity

```
template <typename T>
struct Queue {
    int size = 0;
    ListNode<T> *head = NULL;
    ListNode<T> *tail = NULL;
};
```

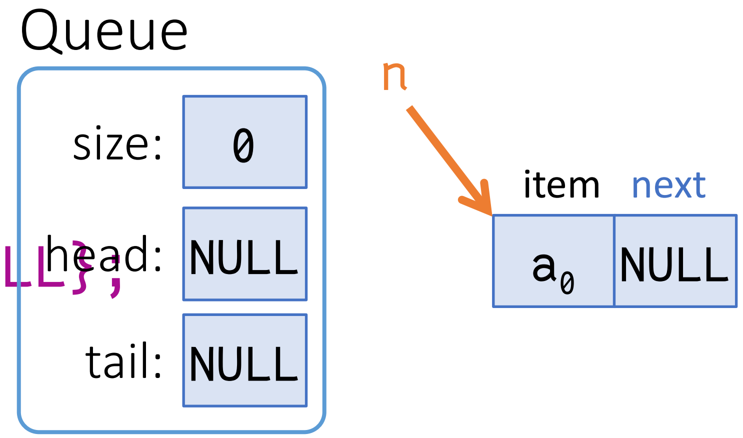
- We can include size and pointer to tail in our struct
- Now enqueue can be done in O(1)

```
queue.tail->next = new_node;
```



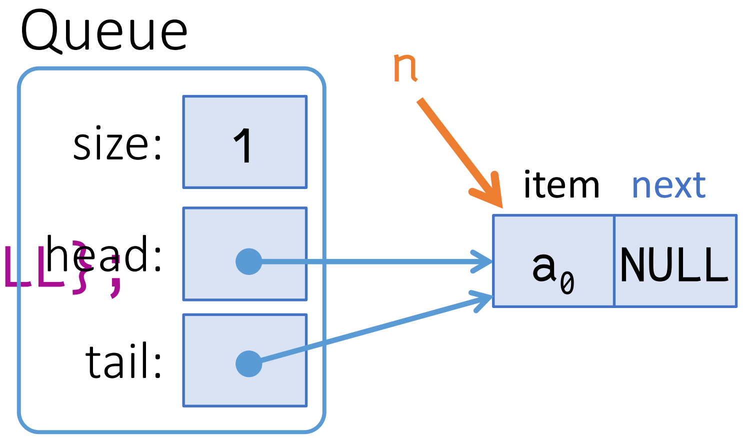
Putting it together

```
template <typename T>
void enqueue(Queue<T> &q, T item) {
  LinkNode<T> *n = new LinkNode<T>{item, NULL};
  if (q.tail == NULL)  // q is empty
    q.head = n;
  else
    q.tail->next = n;
  q.tail = n;
  q.size++;
}
```



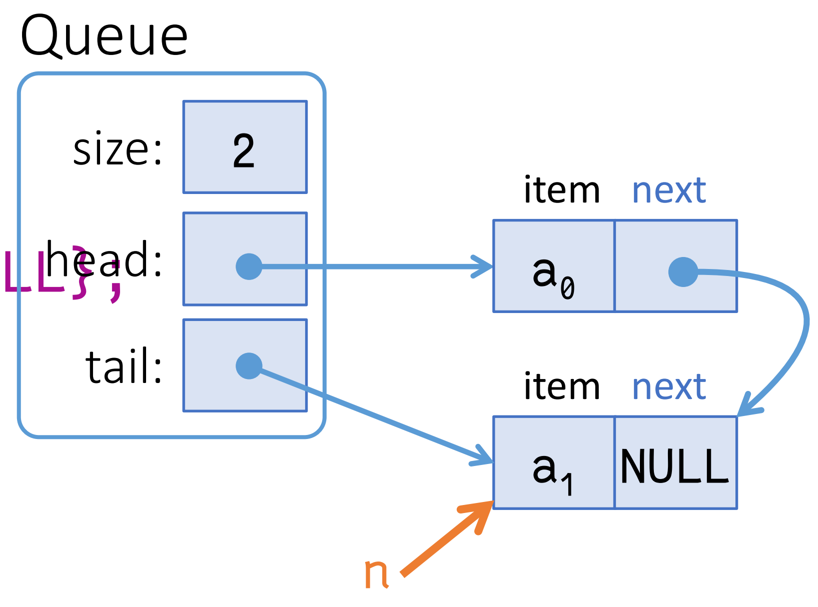
Putting it together

```
template <typename T>
void enqueue(Queue<T> &q, T item) {
  LinkNode<T> *n = new LinkNode<T>{item, NULL};
  if (q.tail == NULL)  // q is empty
    q.head = n;
  else
    q.tail->next = n;
  q.tail = n;
  q.size++;
}
```



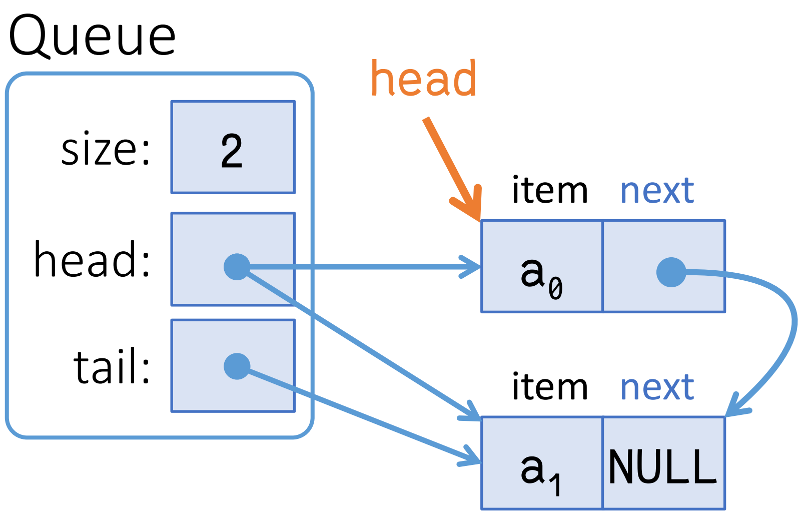
Putting it together

```
template <typename T>
void enqueue(Queue<T> &q, T item) {
  LinkNode<T> *n = new LinkNode<T>{item, NULL};
  if (q.tail == NULL)  // q is empty
    q.head = n;
  else
    q.tail->next = n;
  q.tail = n;
  q.size++;
}
```



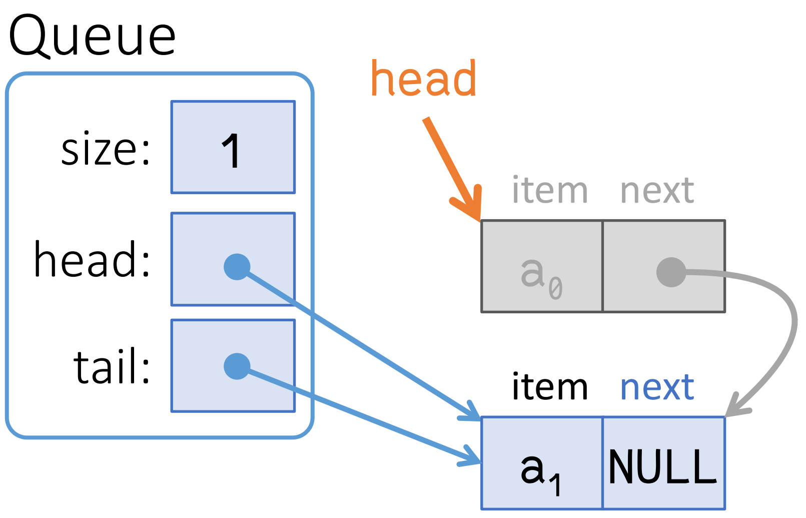
Putting it together

```
template <typename T>
void dequeue(Queue<T> &q) {
  LinkNode<T> *head = q.head;
  if (q.head == q.tail)  // one item in q
    q.tail = NULL;
  q.head = head->next;
  delete head;
  q.size--;
}
```



Putting it together

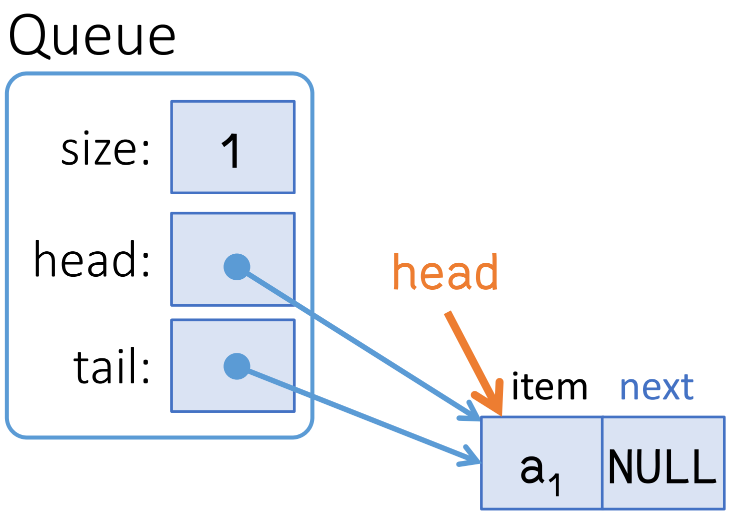
```
template <typename T>
void dequeue(Queue<T> &q) {
  LinkNode<T> *head = q.head;
  if (q.head == q.tail)  // one item in q
    q.tail = NULL;
  q.head = head->next;
  delete head;
  q.size--;
}
```



What happens if you forget to delete?

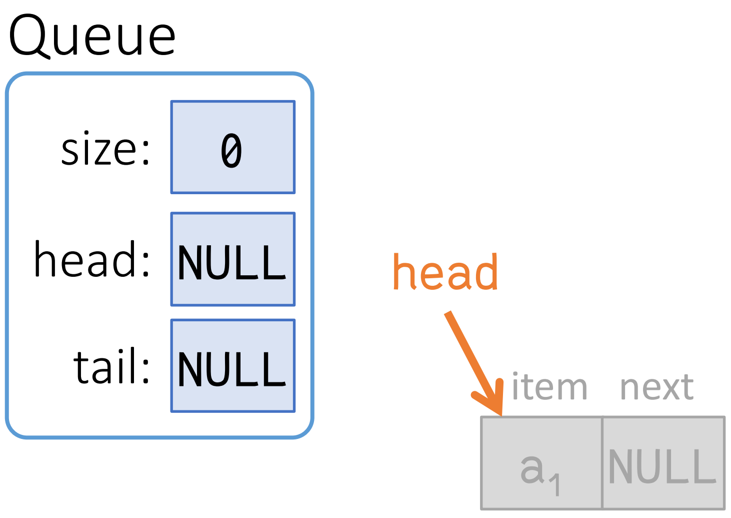
Putting it together

```
template <typename T>
void dequeue(Queue<T> &q) {
  LinkNode<T> *head = q.head;
  if (q.head == q.tail)  // one item in q
    q.tail = NULL;
  q.head = head->next;
  delete head;
  q.size--;
}
```



Putting it together

```
template <typename T>
void dequeue(Queue<T> &q) {
  LinkNode<T> *head = q.head;
  if (q.head == q.tail)  // one item in q
    q.tail = NULL;
  q.head = head->next;
  delete head;
  q.size--;
}
```



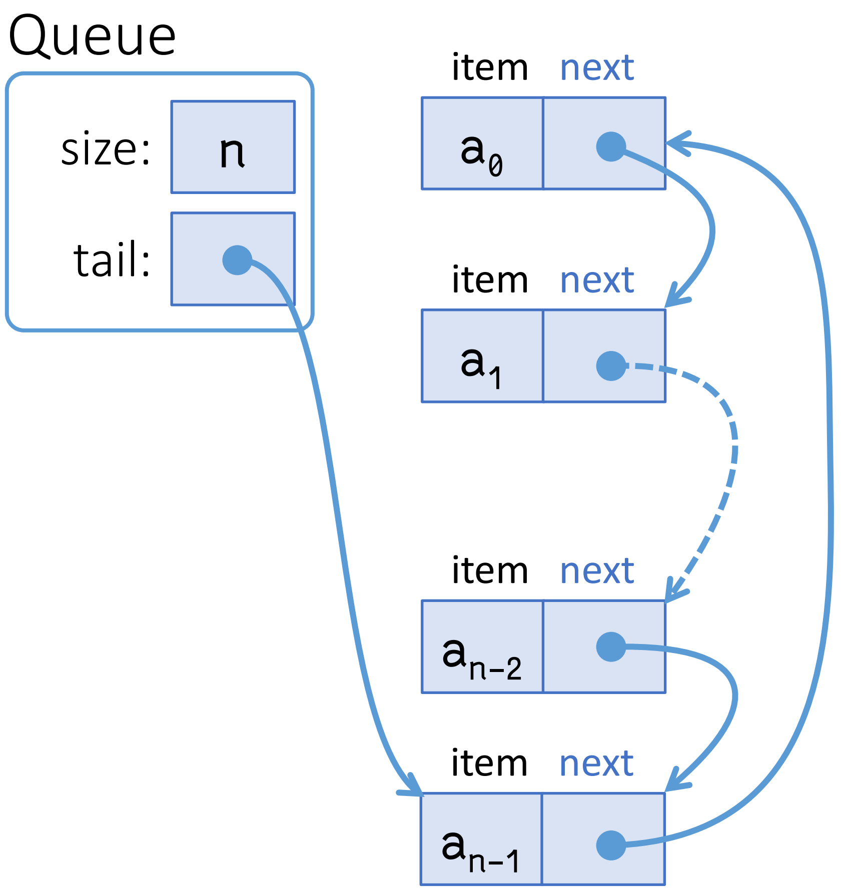
Alternatively

We can use a circular linked list

- Where the tail node links to the head node
- Figure out how to do the enqueue and dequeue operations

More on linked list is TIC2001

- And other data structures



Homemade Queue ADT

1. Using a vector
2. Using two stacks
3. Using a linked list

	Vector	2 stacks	Linked List
Enqueue	$O(1)$	$O(1)$	$O(1)$
Dequeue	$O(n)$	$O(1)$~$O(n)$	$O(1)$

Comparison

Vectors

- Insertion at back: O(1) otherwise O(n) where n is the position
- Deletion same as above
- Accessing any item: O(1) by index

Linked List

- Insertion: O(1) if pointer to node is given
- Deletion same as above
- Accessing any item: O(n) since there is no index

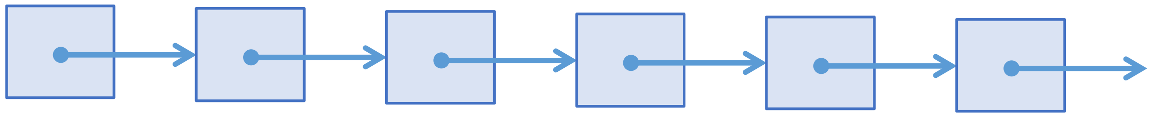
Trees

Angsana, Rain Tree, Yellow Flame, Senegal Mahogany, Tembusu, Sea Almond, Saga, Sea Apple

Another Recursive Structure

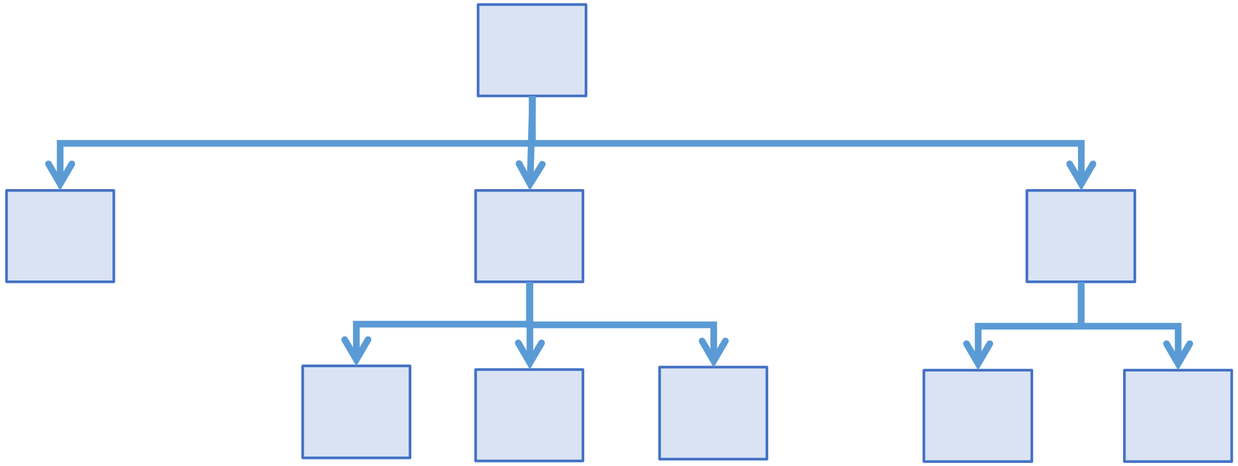
Linked list

– Each node can only link to at most one other node



Tree

– Each node can link to multiple children nodes

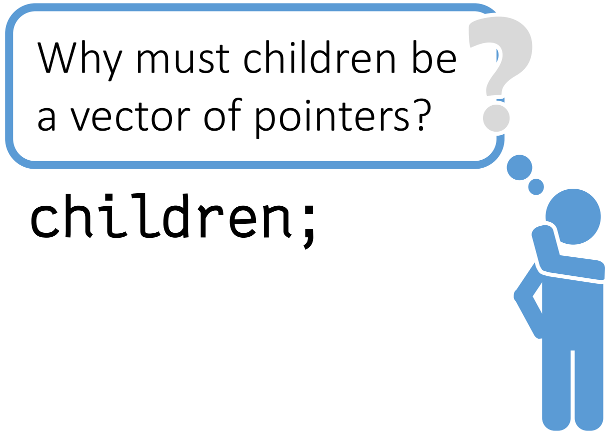


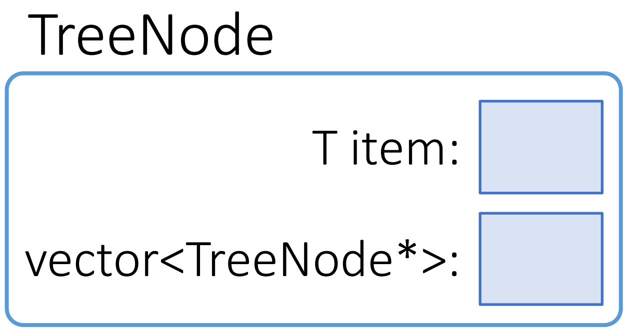
Properties of a Tree

- Each node can have multiple children
  - Nodes with no children are known as leaves
- Each node can have at most one parent
  - Node with no parent is the root
- There can be no cycles in a tree
  - i.e. there must be one and only one root

TreeNode ADT

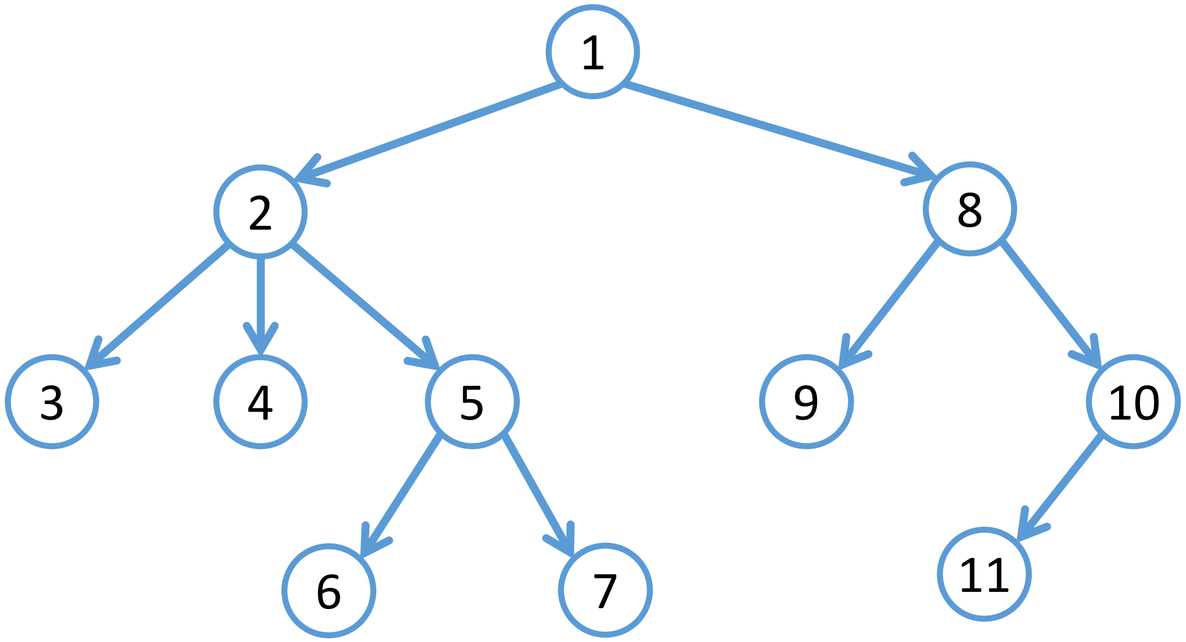
```
template <typename T>
struct TreeNode {
    T item;
    vector<TreeNode*> children;
};

template <typename T>
void add_child(TreeNode<T> &child,
               TreeNode<T> &parent) {
    parent.children.push_back(&child);
}
```





https://goo.gl/MaoWTd

Traversing a Tree



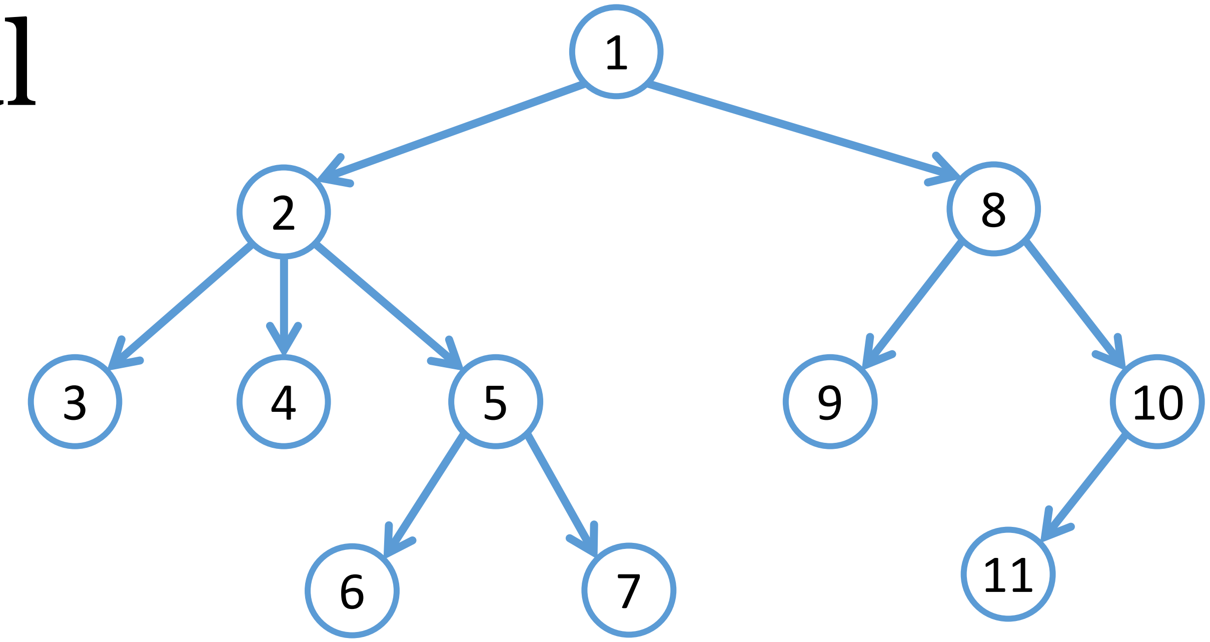
Access all items in the tree

- Searching for an item
- Counting number of items
- Enumerating to a vector, etc.

There are two ways to traverse a tree

- Depth-first → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
- Breath-first → [1, 2, 8, 3, 4, 5, 9, 10, 6, 7, 11]

Depth-first Traversal

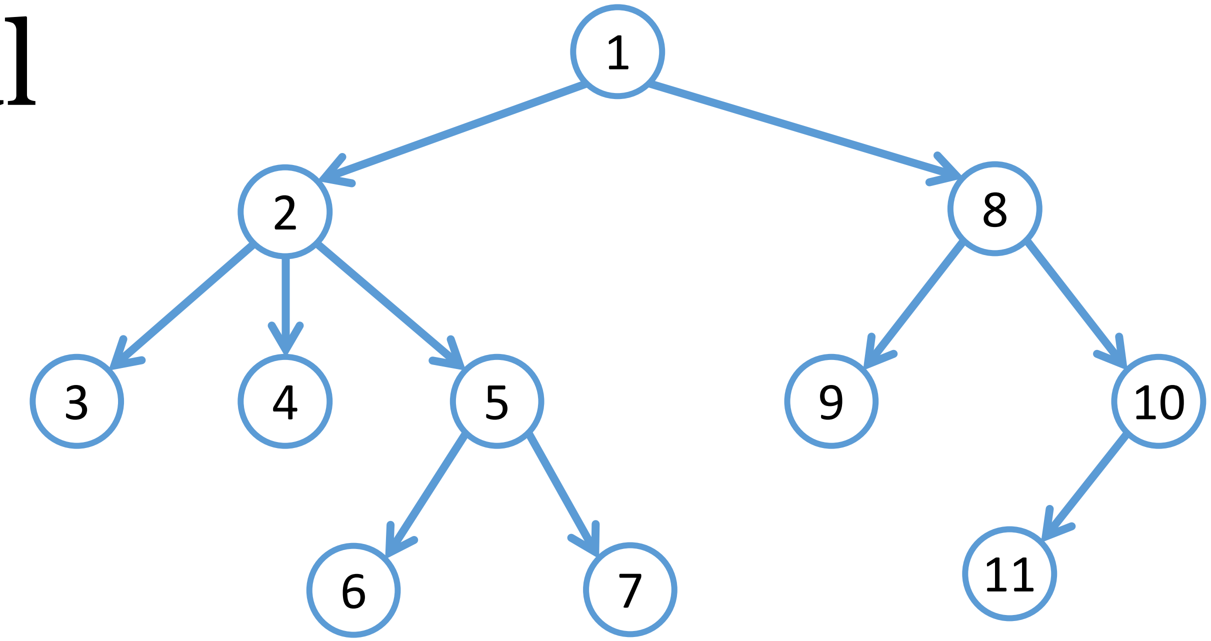


Easily done with recursion

- each child is itself a tree
- same function can be called on each child(

```
template <typename T>
void dft(TreeNode<T> &node) {
    cout << node.item << " ";
    for (int i=0; i < node.children.size(); i++)
        dft(*node.children[i]);
}
```

Depth-first Traversal

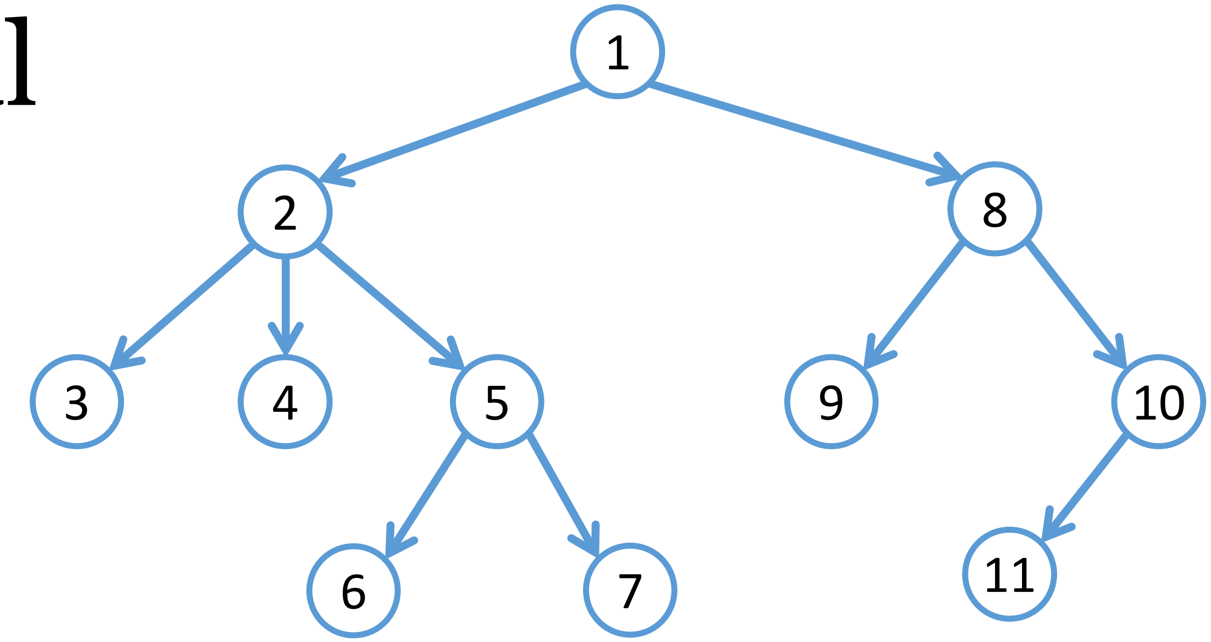


You can either access item

- before (pre-order)

```
template <typename T>
void dft(TreeNode<T> &node) {
    cout << node.item << " ";  // display before
    for (int i=0; i < node.children.size(); i++)
        dft(*node.children[i]);
}
```

Depth-first Traversal



You can either access item

– before (pre-order)
– or after the children (post-order)

```
template <typename T>
void dft(TreeNode<T> &node) {
    for (int i=0; i < node.children.size(); i++)
        dft(*node.children[i]);
    cout << node.item << " ";  // display after
}
```

→ 3 4 6 7 5 2 9 11 10 8 1

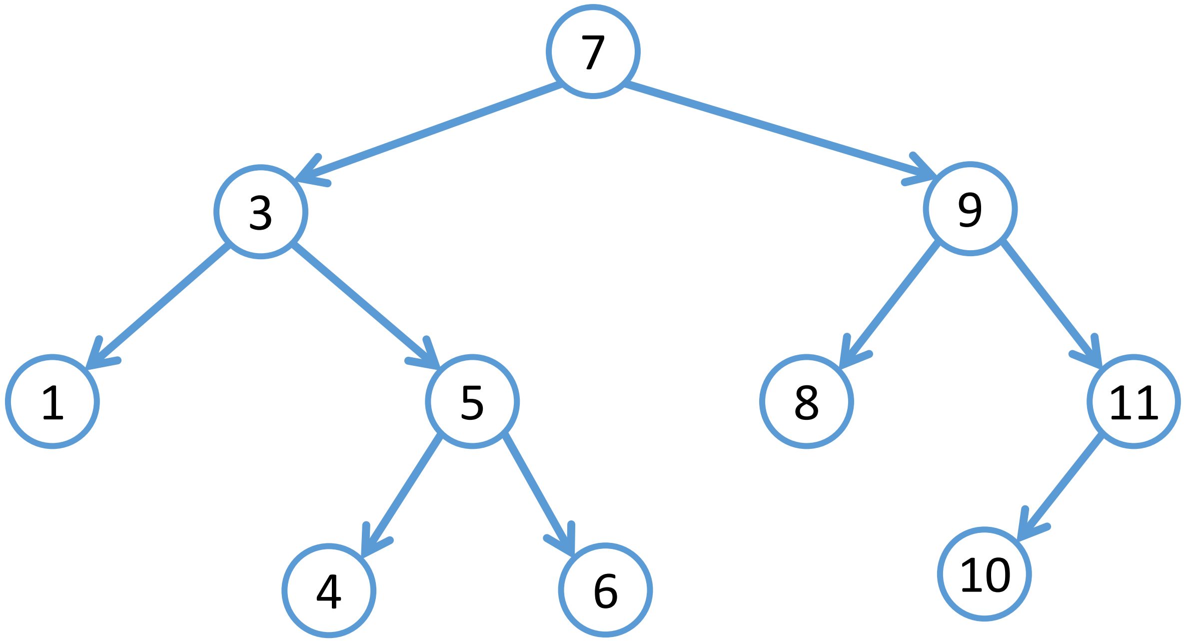
Breath-first Traversal

- Cannot be done recursively
  - At least not without cheating/hacks
  - Requires the use of a Queue
- Will not be discussed in TIC1002
  - More in TIC2001

Binary Search Tree

A special kind of tree

- Each node can have at most two children, i.e. 0, 1 or 2
- Elements are comparable, i.e. can be sorted
- All elements in left subtree of a node is < than its value
- All elements in right subtree of a node is > than its value

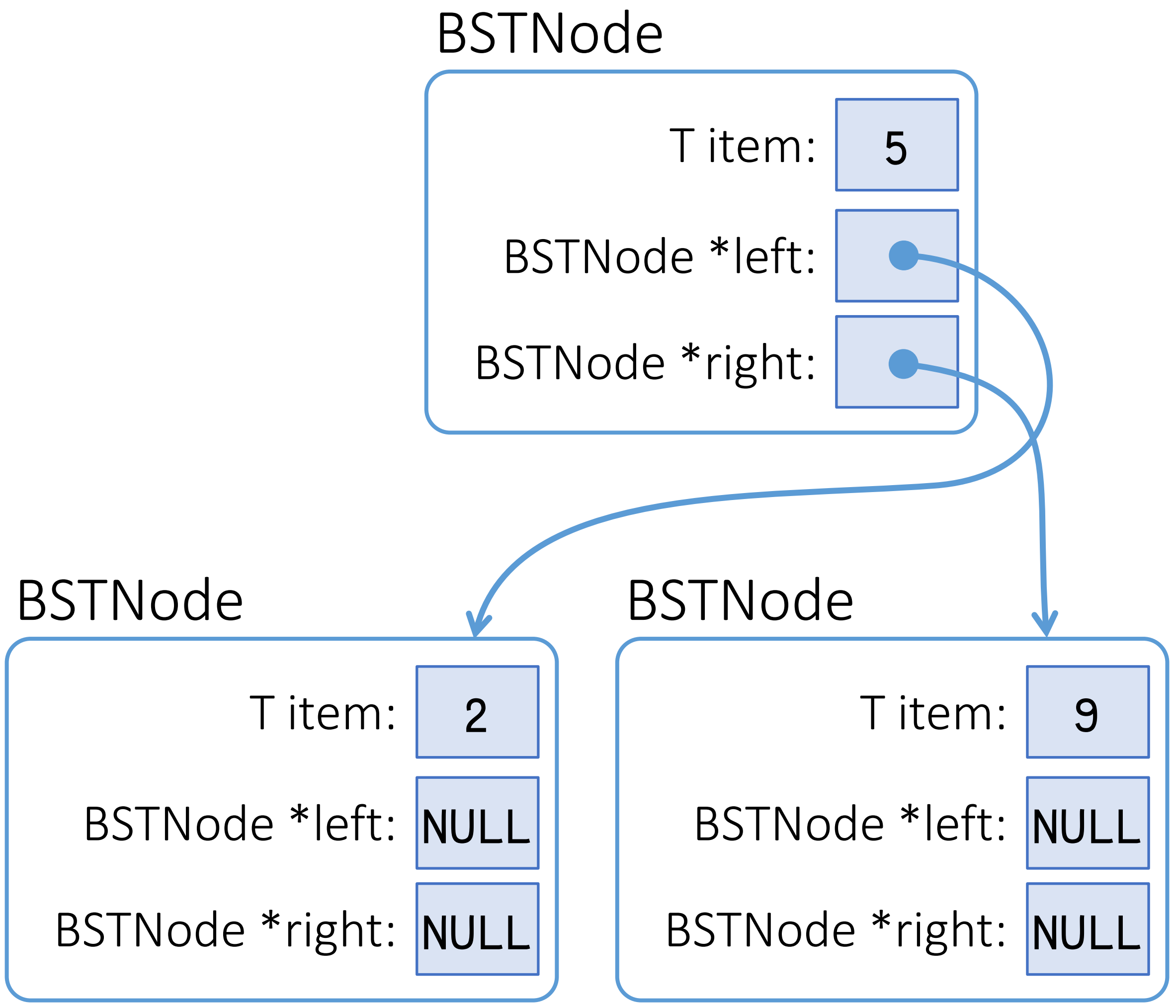


Binary Search Tree

Structure of BST

- We can simplify a TreeNode

```
template <typename T>
struct BSTNode {
    T item;
    BSTNode *left, *right;
};
```



Binary Search Tree

Perform Binary Search

- Key idea: Narrow your search by going left or right
- If search key is < current, search at left child
- If search key is > current, search at right child

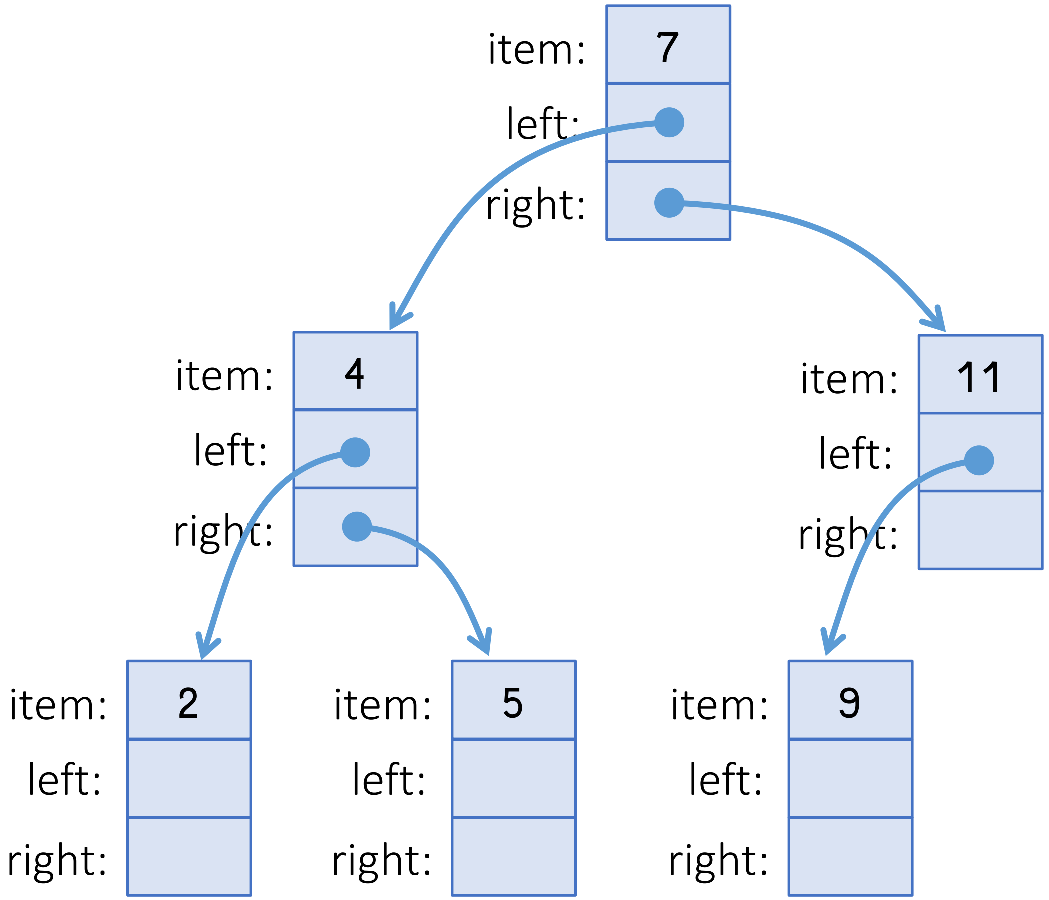
Using recursion

```
bool search(BSTNode<T> *node, T key) {
  if (node == NULL)
    return false;
  if (node->item == key)
    return true;
  if (key < node->item)
    return search(node->left, key);
  else
    return search(node->right, key);
}
```

Binary Search Tree

Try tracing the code

```
bool search(BSTNode<T> *node, T key) {
  if (node == NULL)
    return false;
  if (node->item == key)
    return true;
  if (key < node->item)
    return search(node->left, key);
  else
    return search(node->right, key);
}
```



Summary

Template

- Placeholder
- Actual type declared later

Memory

- Pass-by-value vs Pass-by-reference
- Static allocation on Stack Memory
- Dynamic allocation on Heap Memory

Recursive Containers

- Linked List and Tree
- Dynamically allocate new nodes
- Manual deletion required

- How to search/traverse recursively