Lectures: Data Structures

Programming Systems and Tools

State University of New York at Binghamton

May 2, 2024

BINGHAMTON UNIVERSITY | THOMAS J. WATSON COLLEGE OF ENGINEERING AND APPLIED SCIENCE

Lectures

Vectors

Motivation

Suppose we have a mutable data set with the following requirements:

- Varies in size from 10,000 to 10,000,000 elements
- Requires random access

Can we use a typical C array?

Definition

Vectors: Dynamic Arrays

We will implement `struct Vector` using this definition:

```
1  typedef uint64_t Data;
2  typedef struct Vector {
3    size_t size;     // index of last element in the array
4    size_t capacity; // the total number of elements that can be stored
5    Data *array;     // pointer to the storage space
6  } Vector;
7
8  Vector *newVector(size_t size);
9  void deleteVector(Vector *v);
10 int printVector(const void *v);
11
12 Data *at(Vector *v, size_t index);
13 size_t capacity();
14 void clear(Vector *v);
15 Data *find(Vector *v, Data d);
16 Data *insert(Vector *v, Data d, size_t idx);
17 Vector *resize(Vector *v, size_t count);
18 Data *remove(Vector *v, size_t index);
```


- Dynamic arrays are resized as needed using a predefined algorithm
- Vectors are an *abstract data type*(ADT) implementing the dynamic array
- Functions just like a standard array, except it automatically grows
- Exist in other languages:
  - `std::vector` in C++,
  - `java.util.ArrayList<E>` in Java,
  - `list()` in Python

Initializing Vectors

Vectors: Dynamic Arrays

`Vector *newVector(size_t size);`

1. v ←`(Vector *) malloc(sizeof(Vector))`
2. v.capacity ←size
3. v.size ←0
4. v.array ←`(Data *) malloc(sizeof(Data) *` `v.capacity)`
5. Use `memset` to zero-initialize the array storage
6. return v

Pros

- Allocates memory only as needed
- Allows dynamically-sized arrays
- Resizing penalty can be *amortized*

Cons

- Memory and performance intensive
- Still wastes space, particularly at larger array sizes
- Members must be a single type (not a "bag")

Resizing the Vector

What happens if the size argument is less than v.size?

- Allow the resize, but some data is lost
- Disallow the resize, return a pointer to the original Vector
- Allow `realloc()` default behavior:
  - The contents will be unchanged in the range from the start of the re- gion up to the minimum of the old and new sizes
  - From the minimum size (new capacity) to the maximum size (old capacity), the data is not copied

Inserting into Vector

```
Data *insert(Vector *v, Data d, size_t idx);
```

- There is enough storage space for the index:
  1. v.array[index] ← d
  2. if *index* > *v.len* v.len ← index
- There is not enough storage space for the index:
  1. Resize the array using `resize()`
  2. v.array[index] ← d
  3. if *index* > *v.len*, v.len ← index

Resizing the Vector

```
Vector *resize(Vector *v, size_t count);
```

1. v.array ←`realloc(v.array, v.size * 2 );`
   - If successful (v.array is not NULL):
     1. temp ←v.capacity
     2. v.capacity ←v.capacity * 2
     3. Zero initialize from temp to v.capacity using `memset`
   - If not successful (i.e. `realloc()` fails):
     1. Return a void pointer. It is the caller's responsibility to check this condition.

Resizing the Vector

```
Vector *resize(Vector *v, size_t count);
```

We will eventually need to grow our dynamic array's storage space. Two strategies:

- Every time we add, we *incrementally* expand the array (e.g. if size was 10, we resize to 11, 12, 13, etc.).
  - Keeps array exactly large enough for current needs
  - Growth operations require heap reallocation on every insertion ... no amortization!
- If length equals the size, we *geometrically* increase the size to the next power of two (e.g. if size was 10, we resize to 16, 32, 64, etc.)
  - Array will outsize the current needs, particularly at large array sizes.
  - Growth operations can be amortized

Reallocation in the C Standard Library

```c
#include <stdio.h>  // printf
#include <stdlib.h> // malloc, realloc,free
#include <string.h> // strcpy,strcat
int main () {
  char *str;

  /* Initial memory allocation */
  str = (char *) malloc(6);
  strcpy(str, "Hello");
  printf("String = %s,  Address = %p\n", str, str);

  /* Reallocating memory */
  str = (char *) realloc(str, 14);
  strcat(str, " World!");
  printf("String = %s,  Address = %p\n", str, str);

  /* Undefined freeing */
  str = (char *) realloc(str, 0);
  //free(str);
  return(0);
}
```


Reallocation in the C Standard Library

```c
#include <stdio.h>
void *realloc( void *ptr, size_t new_size );
```

`realloc()` reallocates the given area of memory.

- It must be previously allocated by malloc(), calloc() or realloc() and not yet freed with a call to free() or realloc().
- Reallocation happens by one of two strategies:
  - Expanding or contracting the existing area pointed to by ptr, if possible. The contents of the area remain unchanged up to the lesser of the new and old sizes. If the area is expanded, the contents of the new part of the array are undefined.
  - Allocating a new memory block of size new_size bytes, copying memory area with size equal the lesser of the new and the old sizes, and freeing the old block.
- If there is not enough memory, the old memory block is not freed and null pointer is returned.

Reading from a Vector

```
Data *at(Vector *v, size_t index);
```

Reading from vectors is just like using the indexing operator on arrays, except:

- Instead of returning the actual value, we return a pointer; this allows us to have a failure state...
- We must check bounds: if the index is larger than the bounds of the vector, return NULL
- Is it a problem to have an index less than the capacity, but larger than the size?
  - → *No!* ... We're not causing a buffer overread (the memory's allocated, right?)
  - → *Yes!* ... We're violating the intent of the data structure
  - → **We'll consider it a problem and return a nullptr.**

Experiment: Using realloc()

1. Create a file **realloc.c**
2. Allocate a 6 byte-wide character buffer on the heap using `malloc()`.
3. Use `strcpy()` to copy the string "Hello" into the buffer
4. Print the address of the buffer, and the string in the buffer.
5. Use `realloc()` to change the size of the original buffer to 14 bytes.
6. Use `strcat()` to append the string " World!" to the original buffer.
7. Again, print the address of the buffer and the string in the buffer.
8. Call `realloc()` a second time, this time with size 0. Do not call free.
9. Compile the program and run it in Valgrind. Is there a memory leak?

Clearing a Vector

```
void clear(Vector *v);
```

Clearing the vector is just iterating across the entire vector, and removing the value at each index:

1. For index i in range 0 ... v.size - 1: v.array[i] ←0
2. v.size = 0

- Alternative option: memset from index 0 to size . . . Much faster!
- Should we resize the array also? **We do not. Why?**
  - Relies on memory allocator not causing a failure
  - Expensive to call realloc()
  - Don't quietly change state on the user.

Removing from a Vector

```
Data *remove(Vector *v, size_t index);
```

1. Actual deletion by deallocation of that part of vector memory
   - Expensive and must reallocate space carefully
   - What if item deleted is not last item?
2. Set v.array[index] to an "empty" value
   - Most efficient option
   - Doesn't cause problems with memory management or external references
   - Need to reserve an "empty" value ... no "None" value in C
3. Remove the item and move items at higher indicies up to size-1.
   - Most accurately models behavior of bounds checked buffers
   - Could cause expensive iteration to move items leftward!
   - Dangerous to mutate storage while also iterating across it!

Removing from a Vector

What if another variable outside the struct has a reference for an index into the array?

- Vector cannot stop unauthorized access into the array
- Unauthorized access might not cause `.size` or `.capacity` to be properly updated

Lists

Vectors Hide Underlying Complexity

- Suppose we have a dynamic vector v with `.array={1,4,10,19,6}, .capacity=5`.
- Then suppose we want to call `insert(v,7,2)`. What vector operations have to happen?
  1. `resize(v,8)`:
     1. Traverse heap free list to find a new array storage: $O(n)$
     2. Acquire new heap storage: $O(1)$
     3. Copy array contents from old heap storage to new: $O(n)$
     4. Deallocate old heap storage: $O(1)$
     5. Update v.capacity = 8: $O(1)$
  2. Copy 3 array items to next index: $O(n)$
  3. Insert 7 at index 2: $O(1)$
  4. Update v.size = 6: $O(1)$

Motivation

Vectors have problems

C dynamic vectors are:

- **Inflexible**: Vectors store in statically sized arrays, meaning once the size is allocated the original array cannot change.
- **Wasteful**: Over-allocation happens "in case you need it"
- **Cumbersome**: Insertion requires resizing, resizing requires reallocation.

Linked Lists Remove Underlying Complexity

- The problem with arrays is that they must be contiguous in memory and statically sized.
  - Resizing is entirely dependent on the current state of memory.
- What if we could link non-contiguous segments of memory?
  - Insertion and deletion are less cumbersome – we just link to the subsequent or previous segment.
  - Resizing, inserting or deleting are a matter of changing links between segments.

Vectors Hide Underlying Complexity

- Suppose we have a dynamic vector v with `.array={1,4,10,19,6}, .capacity=5`.
- Then suppose we want to call `insert(v,7,2)`. What vector operations have to happen?

What Goes in the Node?

Allowing mixed types in a List

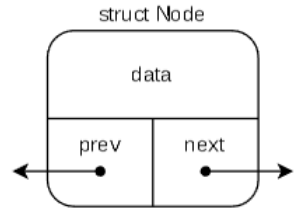
```c
#include <stdint.h>

/* typedef option */
typedef uint64_t Data;

/* payload struct option */
typedef struct Data {
  union {
  uint64_t uint64;
  int64_t int64;
  } data;
  enum type { UINT64, INT64 };
} Data;
```


Node Objects

- We will create our list using objects called **nodes**.
- Each node represents one piece of linked data and is comprised of two or three members:
  - Its data payload, and either:
  - A pointer to the next node only (*singly linked*)
  - A pointer to both the previous and next nodes (*doubly linked*)
- Every node contains the address of the next node and previous node with two exceptions:
  - The **head** contains `null` in its previous pointer.
  - The **tail** contains `null` in its next pointer.




```c
typedef struct Node{
  Data data;
  struct Node *next;
  struct Node *prev;
} Node;
```


What Goes in the Node?

Pros of typedefs

- Easy to implement
- Single point to change

Cons of typedefs

- Every Data member has to be same type throughout the program
- Must keep uses of Data synchronized to type

Pros of struct Data

- We can have multiple payload types throughout the program
- We can have mixed payload types, even within the same data structure!

Cons of struct Data

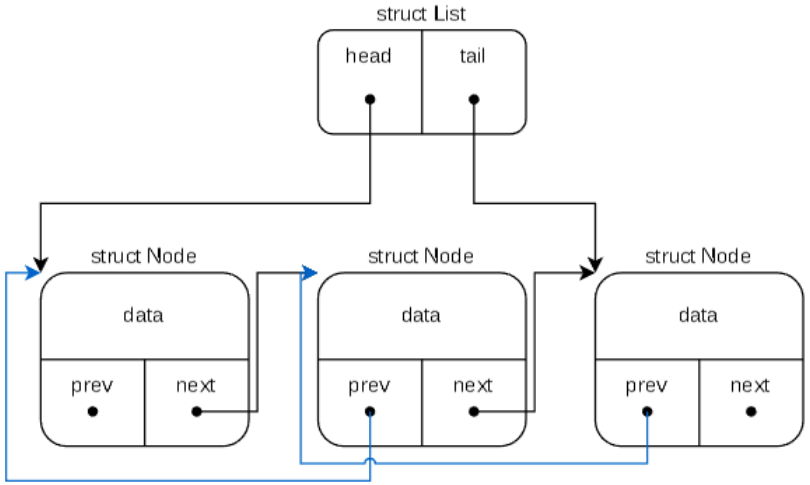
- Harder to implement
- Uses more storage space
- Must access the payload through the union
- Must keep Data's enumeration in sync

What Goes in the Node?

Allowing mixed types in a List

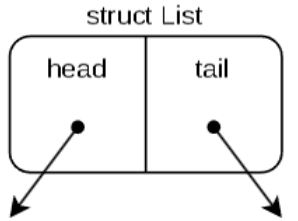
- No matter what kind of data is stored in the Data member, the algorithms for our `List`'s "CRUD" (Create, Read, Update, Delete) operations will be the same.
- In other languages, we can use *templates* to hold the data, making the data structure type agnostic.
- In C, we *could* make our `Nodes` able to hold multiple types of data by typedefs or payload structs

Doubly Linked List



List Objects

- A linked list object's only *required* member is a pointer to the head of the list
- For a doubly-linked list, we can add a pointer to the tail for convenience, and a count of the number of linked `Nodes` in the `List`
- For an empty list, we set the head (and tail) pointers to NULL, and the size to 0
- Do not use the head or tail pointers to traverse the list. Copy them into another pointer first. ... **Why?**
- We don't include any data payload in the list object itself ... **Why?**



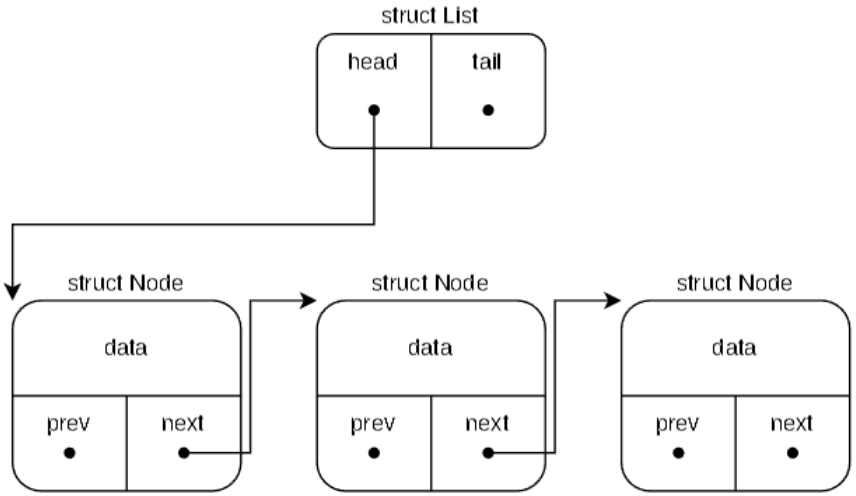

```
1  typedef struct List {
2    Node *head,
3    Node *tail;
4    size_t size;
5  } List;
```


List Object Definition

We will implement the List with the following definition:

```
1  #ifndef DATA
2  #define DATA
3  typedef uint64_t Data;
4  #endif // DATA
5
6  #ifndef LIST_H
7  #define LIST_H
8
9  typedef struct Node {
10   Data data;
11   struct Node *next, *prev;
12 } Node;
13
14 typedef struct List {
15   Node *head, *tail;
16   size_t size;
17 } List;
18
```


Singly Linked List



List Object Definition

```
1  Node *newNode(Data d);
2  void deleteNode(Node *n);
3  int printNode(const void *n);
4
5  List *newList();
6  void deleteList(List *l);
7  int printList(const void *l);
8
9  List *append(List *l, Node *n);
10 void clear(List *l);
11 Node *find(List *l, Data d);
12 List *insert(List *l, Node *n, Node *pos);
13 size_t length(List *l);
14 List *remove(List *l, Node *pos);
15 void traverse(List *l, int (*func)(const void *));
```

All of the object management functions work as with Vectors:

- The `new` function returns a pointer to the newly created object
- The `delete` function frees the memory of the object pointed to by the argument. In the case of the `deleteList()` function, it must also delete all the Nodes in the List first.

Copying a Linked List

What happens when we use the assignment operator with two pointers to List objects? `struct List new_list = old_list;`

Shallow Copy Only copy the pointer address. Both list pointers point to the same list

Deep Copy Copy the sub-objects from the old object into the new object, including their values. Each list pointer points to one of two identical but separate lists

C will do a shallow copy with the assignment operator. We must implement a `List *copyList(List *l)` function to get deep copy through list traversal ... This is an exercise to the reader.

Getting the length of a List

```
size_t length(List *l);
```

Two options:

- Traverse the `List`, counting the number of nodes we encounter until we reach either a NULL pointer or the tail node: $O(n)$
- Update the `List.size` field after each `append()`, `clear()`, `insert()`, `remove()` operation: $O(1)$ amortized

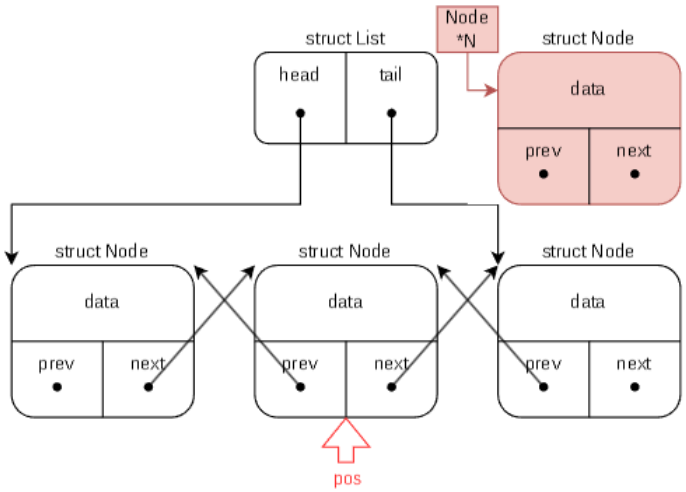
Copying a Linked List

What happens when we use the assignment operator with two pointers to List objects? `struct List new_list = old_list;`

Shallow Copy Only copy the pointer address. Both list pointers point to the same list

Deep Copy Copy the sub-objects from the old object into the new object, including their values. Each list pointer points to one of two identical but separate lists

Adding Nodes to the List



Getting a Node from the List

```
Node *find(List *l, Data d);
```

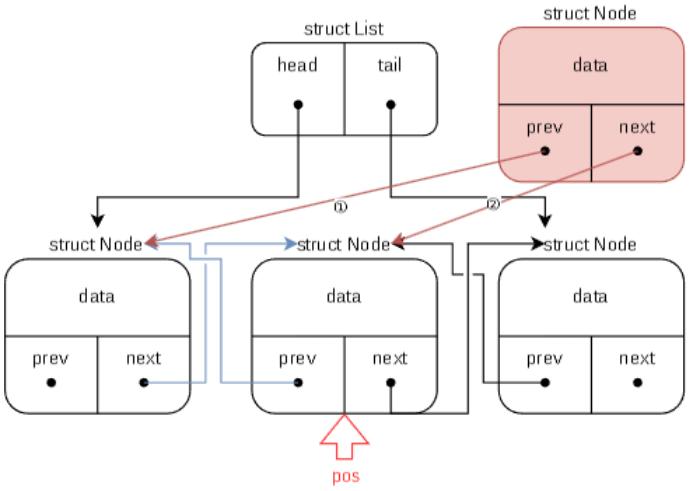
1. Create an iterator pointer and initialize it to `l->head`
2. Check the contents of the node for a match,
   - if matched, return
   - If no match, advance the iterator to `iter->next`
3. If `iter == NULL`, break and return

Return values:

- We found the first node with the desired contents: return a pointer to that node
- We go past the tail of the list to a `nullptr`: we return a NULL pointer to indicate we couldn't find the right node

→ Either way, `iter` points to the value we want to return!

Adding Nodes to the List

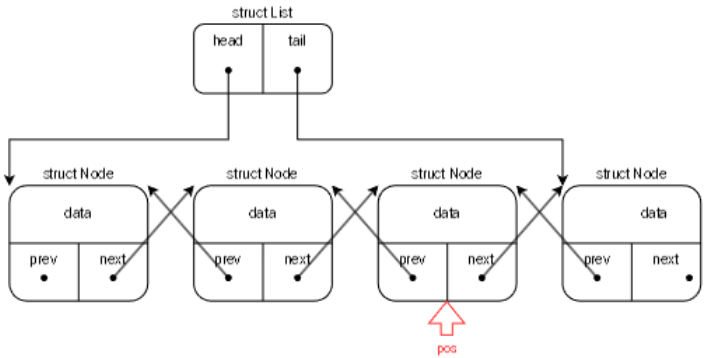


Adding Nodes to the List

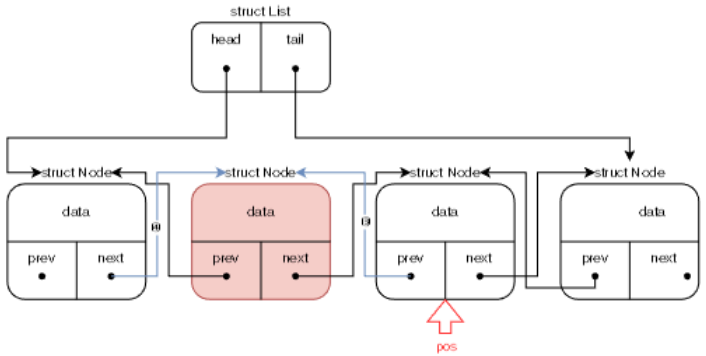
```
List *insert(List *l, Node *n, Node *pos);
List *append(List *l, Node *n);
```

- The difference between the two functions is that `*insert_node(List *l, Node *n, Node *pos)` inserts `n` before `pos`. `List *append_Node(List *l, Node *n)` inserts the node after the last node in the List (i.e. at a new `l->tail`).
- When inserting, connect the new node to the list before updating pointers already in the lists' nodes.
- Once the new node is properly connected, it can be used to update the existing `List` node pointers.
- Failure to properly connect the new node will cause a memory leak due to lost nodes, and/or segmentation faults when traversing the list.

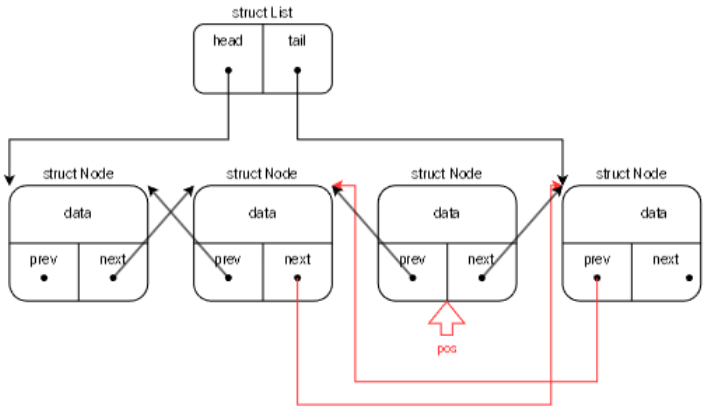
Removing Nodes from the List



Adding Nodes to the List



Removing Nodes from the List



Removing Nodes from the List

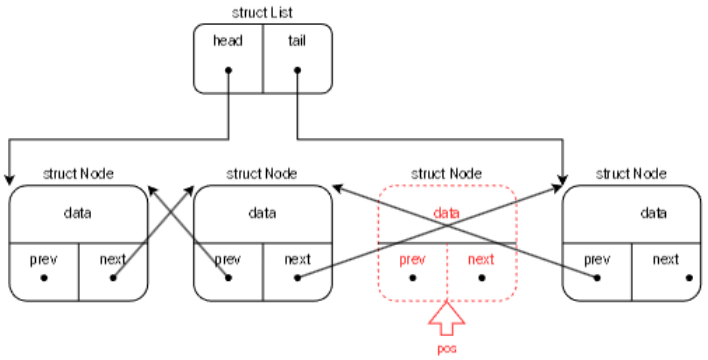
```
List *remove(List *l, Node *pos);
```

1. Find the node before `pos` if singly-linked (the predecessor, `pred`). The node after (the successor, `succ`) can be found using `pos` itself.
2. Update `pred->next` to point to successor, `succ->prev` to point to predecessor.
3. Delete the node pointed to by `pos` to prevent a memory leak and return a pointer to the newly updated list.

Abstract Data Types

- Data Structures (DS) are the implementation that manage data
- Abstract Data Types (ADT) describe behavior, not implementation
  - A DS is an implementation of an ADT
  - DS should have a standardized interface, but the underlying implementation can vary greatly

Removing Nodes from the List



Stacks

Stacks are a sequential collection:

Sequential Order of element insertion matters

Collection Contains items of the same type, forming a grouping

- The purpose of a Stack ADT is to describe how the data is accessed, not how the data is stored!
- Two popular Stack DS implementations are linked list and array
- The Stack DS implementations provide the ADT's sequential access control through encapsulation

Stacks and Queues

Stack Access Control

- Stacks are **Last In, First Out** (LIFO)
- Items are removed in the reverse order of their insertion.
- Think of stacks as being like a pile of plates: to get to a lower plate, we have to lift the plates above it.
- Stacks limit the user to accessing only the newest element.

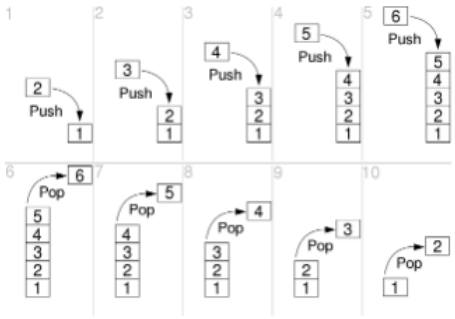
Examples of Stacks

What are some examples of stacks in computer science?

- Function call stack
- Exception handling/unwinding
- Undo/Redo commands
- Syntax checking - are brackets properly matched?
- Prefix notation calculators ("RPN")

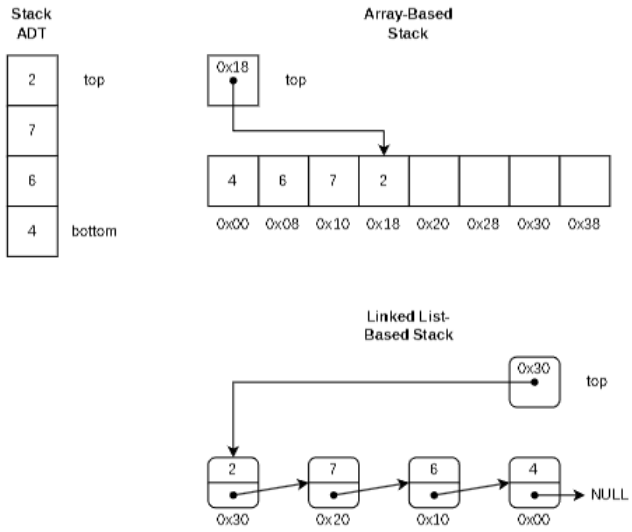
Stack Update Operations

- Push
  - Inserting an item onto the stack
  - Imagine pushing down on everything already on the stack, one place farther from the top
- Pop
  - Removing from the stack
  - Each element remaining on the stack moves one place closer to the top
  - To access an element below, you must pop every element above it
  - `top()` or `peek()` get the top item like `pop()` but without removing it from the stack



Examples of Stacks

What are some examples of stacks in computer science?



Array Based

- 'top' is index of the most recently appended item
- Top of the stack is at the higher address than the bottom

Pros of Array Based

- Easier memory management

Cons of Array Based

- Must have a static, maximum size or a dynamic vector storage


```
1  #ifndef DATA_T
2  #define DATA_T
3  typedef uint64_t Data;
4  #endif // DATA_T
5
6  #ifndef STACK_H
7  #define STACK_H
8
9  typedef struct Node {
10   Data value;
11   struct Node *next;
12 } Node;
13
14 typedef struct Stack {
15   Node *top;
16   size_t size;
17 } Stack;
```


Linked List Based

- 'top' is a pointer to the head of the list
- Stack grows at the head of the list

Pros of Linked List Based

- Internal storage is dynamically allocated → always right-sized

Cons of Linked List Based

- Push and pop have increased memory management

Queue Access Control

- Queues are **First In, First Out** (FIFO)
- Items are removed in the order of their insertion.
- Think of queues as being like a line at the airport: each element must wait until all other elements that arrived first are processed.
- Queues limit the user to accessing only the oldest element.

Stack Interface

```
1  // Not shown: new, delete, print methods
2  void clear(Stack *s);
3  size_t size(Stack *l);
4  Node *top(Stack *s);
5  Node *pop(Stack *s);
6  Node *push(Stack *s, Data d);
7  void traverse(Stack *s, int (*func)(const void *));
```


Examples of Queues

What are some examples of queues in computer science?

Queues

Like stacks, queues are also a sequential collection:

Sequential Order of element insertion matters

Collection Contains items of the same type, forming a grouping

- Like stacks, queues provide restricted access control through encapsulation
- Unlike stacks, queues are First In First Out (FIFO)

Queue Implementations

Array Based

- 'head' and 'tail' are pointers to the front and end of the queue in the array
- Head of the queue is at a lower address than the tail of the queue

Pros of Array Based

- Easier memory management

Cons of Array Based

- Requires array to have a static, maximum size or a dynamic vector

Examples of Queues

What are some examples of queues in computer science?

- I/O buffers
- Operating system pipes
- Multithreading synchronization primitives
- First Come, First Serve (FCFS) process scheduling
- Cryptographic block and stream ciphers

Queue Implementations

Linked List Based

- 'head' is a pointer to the head of the list, 'tail' is a pointer to the end of the list
- Queue grows at the tail of the list

Pros of Linked List Based

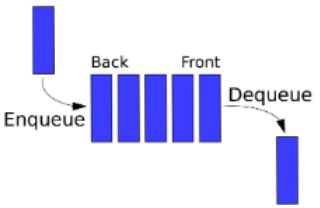
- Internal storage is dynamically allocated → always right-sized

Cons of Linked List Based

- Enqueue/Dequeue have increased complexity

Queue Update Operations

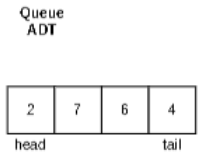
- Enqueue
  - Inserting an item into the queue
  - Imagine adding something to the end of the line. No other items' position needs to change as long as the queue grows to the back.
- Dequeue
  - Removing an item from the queue
  - Each element remaining in the queue moves one place closer to the front
  - To access an element behind, you must dequeue every element inserted before it
  - `front()` or `peek()` get the front item like `dequeue()` but without removing it from the queue

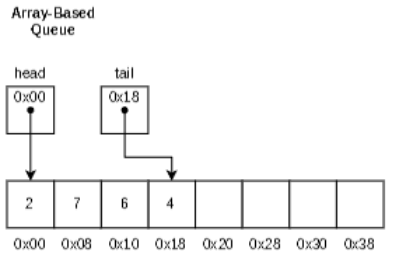


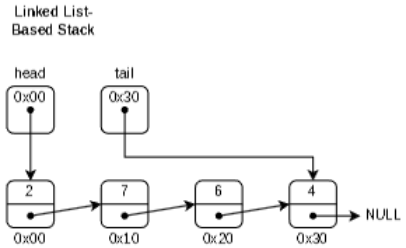
Queue Interface

```
1  /* Not shown: new, delete, print methods */
2  Node *back(Queue *q);
3  void clear(Queue *q);
4  Node *dequeue(Queue *q);
5  Node *enqueue(Queue *q, Data d);
6  Node front(Queue *q);
7  size_t size(Queue *q);
8  void traverse(Queue *q, int (*func)(const void *));
```


Queue Implementations







Circular Buffer

A specialized implementation of queues, using a single, fixed-size buffer with wraparound addressing as though the buffer was connected end-to-end. Like queues, circular buffers are also a sequential collection:

Sequential Order of element insertion matters

Collection Contains items of the same type, forming a grouping

- Like queues, circular buffers provide restricted access control through encapsulation
- Like queues, circular buffers are well-suited to be used for First In First Out (FIFO) access control, except they have the advantage of not needing any memory resizing or copying (all operations are constant time).

Queue Interface

```
1  #ifndef DATA_T
2  #define DATA_T
3  typedef uint64_t Data;
4  #endif // DATA_T
5
6  #ifndef QUEUE_H
7  #define QUEUE_H
8
9  typedef struct Node {
10   Data value;
11   struct Node *next;
12 } Node;
13
14 typedef struct Queue {
15   Node *head, *tail;
16   size_t size;
17 } Queue;
```

Examples of Circular Buffers

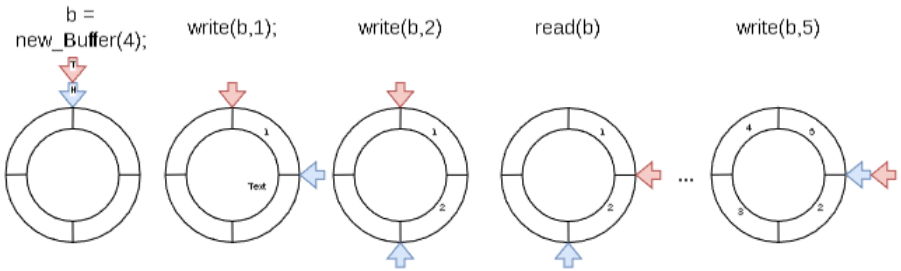
What are some examples of circular buffers in computer science?

- Keyboard input
- Producer-consumer pattern

Circular Buffer Access Control

- Circular buffers are **First In, First Out** (FIFO)
- Items are removed in the order of their insertion.
- Think of circular buffers being like clocks, where the hands point to the item in storage to be read, or the location in storage to write the next item.

Circular Buffer Implementation



Write:

1. buffer[tail] ←value
2. tail ← *tail* + 1 mod *capacity*

Remove:

1. temp ← buffer[tail]
2. tail ← *tail* − 1 mod *capacity*
3. return temp

Read:

1. temp ← buffer[head]
2. head ← *head* + 1 mod *capacity*
3. return temp

Examples of Circular Buffers

What are some examples of circular buffers in computer science?

Deque Implementations

- Deques can be implemented with different types of storage:
  - head-tail linked list (most common)
  - dynamic array (sometimes implemented as a circular array when maximum storage size is known)
- Deques can be implemented with different access restrictions:
  - Insertion and deletion at both ends (most common)
  - Input restricted: deletion from both ends, but insertion at one end only
  - Output restricted: insertion at both ends, but deletion at one end only

Circular Buffer Interface

```c
#include <stdint.h>
#include <stddef.h>

typedef uint64_t Data;
typedef struct Buffer {
  size_t *head,*tail;
  Data* storage;
  size_t size;
}

// Not shown:
// new, copy, del methods for Buffer
void clear(Buffer *b);
void write(Buffer *b, Data d);
Data read(Buffer *b);
```


Examples of Deques

What are some examples of deques in computer science?

Double-Ended Queue

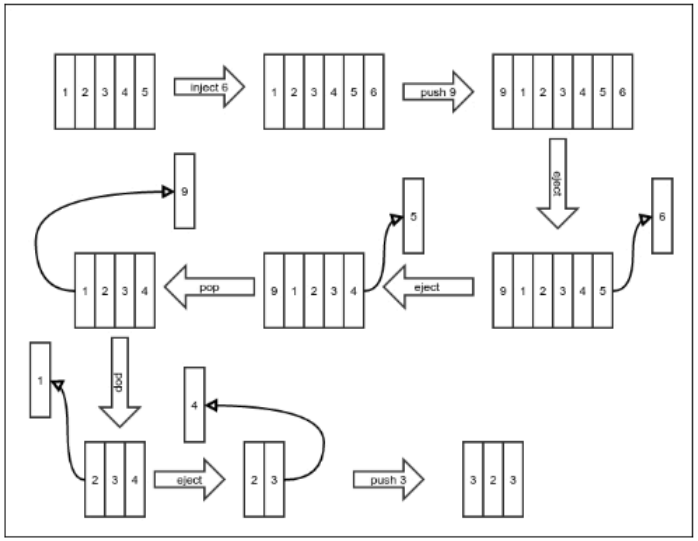
A **double-ended queue** or **deque** (pronounced *deck*) is a generalization of the queue plus the stack. Deques are still sequential collections:

Sequential Insertion direction and timing creates an ordering

Collection Contains items of the same type, forming a grouping

- Deques provide restricted access control through encapsulation
- Insertion and removal can happen at either side of the storage, but not in the middle of the storage.

Deque Operation



Examples of Deques

What are some examples of deques in computer science?

- Scheduling with "work stealing" for multiprocessor computers and parallel programming (A-steal algorithm)
- "Sliding window" problems
- Browser history with bounded size (LIFO) plus forward and back buttons (FIFO)

Trees

Deque Operation

The basic operations of the deque are enqueue and dequeue at either end:

Enqueue:

- Insert element at back:
  - inject (textbooks)
  - push back (C-like languages)
  - . . . or append, snoc
- Insert element at front:
  - push (textbooks)
  - push front (C-like languages)
  - . . . or prepend, unshift, cons

Dequeue:

- Remove element at back:
  - eject (textbooks)
  - pop back (C-like languages)
  - . . . or delete
- Remove element at front:
  - pop (textbooks)
  - pop front (C-like languages)
  - . . . or shift
- Deque implementations also sometimes offer a "peek" that returns the value without dequeing it.

Properties of a Node

key The value stored in a node.

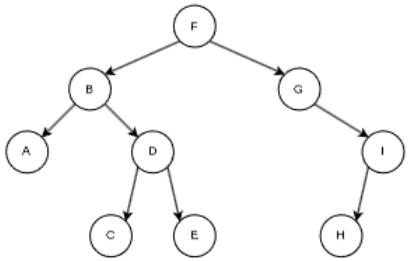
depth The distance of branches from a node to the root of its tree. For a root node, $depth = 0$.

level The set of nodes which have the same depth. The root node alone is $level = 0$

degree The number of children. For a leaf node, $degree = 0$.

Questions:

- What are the depths of A, C?
- What nodes are in level 3?
- What are the degrees of F,G, I and H?



Can We Improve Our Existing ADTs?

- Vectors make working with arrays easier, but no performance or storage improvements
- Linked List improves the array, but slow on random access
- How can we improve?
  - Using linked nodes gives us memory flexibility
  - Add additional node pointers to each node
  - Use divide-and-conquer *recursive* algorithms to make access time grow logarithmically

Properties of a Node

Root Node The node with a depth of zero.

Leaf Node A node with no children.

Internal Node A node with one or more children.

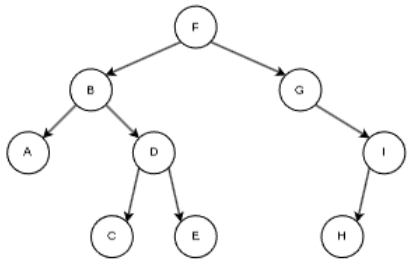
Sibling Node A node which shares a parent.

Ancestor Node All node(s) which are on the path between a node and the root.

Descendent Node All node(s) in a parent node's subtree.

True or False?

- Every tree has a single root node.
- Every non-empty tree has at least one leaf node.
- A root node can be a leaf node or an internal node, but not both.
- For any subtree, all nodes on a level have the same ancestors.



Definition of a Tree

Tree

A finite set of connected nodes such that one node is designated as the root, and all other nodes are recursively partitioned into sets called *subtrees*. Nodes are connected to one another by edges called *branches*.

Hierarchical Relationships between nodes descend from *root* node to *leaf* nodes. There is exactly one path of *branches* from the root to each node. Trees cannot be cyclic.

Connected Each node except *root* must be connected to exactly one *parent*, but may be connected to zero or more *children*. Trees cannot have more than one root.

Binary Search Tree Definition

Binary Search Tree (BST)

A tree with the additional properties:

- The tree's nodes have a *degree* ≤ 2
- Every parent node's left subtree has keys strictly less than the node's key, and the right subtree has keys strictly greater than the parent node's key.
- All node keys are unique.

Properties of a Tree

Size The number of nodes in the tree.

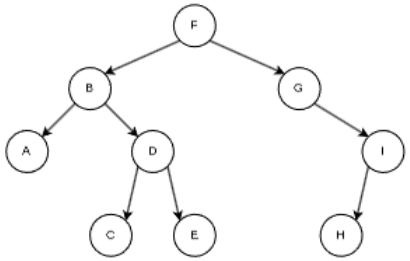
Degree The maximum degree of any node in the tree.

Height The maximum depth of all nodes in the tree.

Width The number of nodes in a level.

Breadth The number of leaves in the tree.

- What is the size, degree, height, and breadth of this tree?
- Which level(s) have maximum width?



BST Implementation

Array

- $index_{left} = 2(index_{parent}) + 1$
- $index_{right} = 2(index_{parent}) + 2$
- $index_{parent} = \lfloor \frac{index_{child}-1}{2} \rfloor$

Pros

- $O(1)$ access, $O(n)$ storage

Cons

- Tree's max size must be known, or the array must be resized.
- How to represent empty nodes?

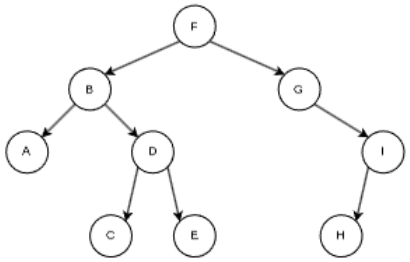
Properties of a Tree

Full Every node has either a zero or the maximum degree

Complete All levels are full, except for possibly the last level, and the nodes in that level are as far left as possible.

Balanced A tree whose left and right subtrees' heights differ by not more than one level.

- Is this tree full?
- Is this tree complete?
- Is this tree balanced?



Recursion

$$f(n) = \begin{cases} \text{atomic object} & \text{base case} \\ f(n') & \text{otherwise} \end{cases}$$

- Recursion is process of a function calling itself to iterate → Use the stack as loop
- **How does it work?**
  1. Determine *base case* that ends recursion and produces an *atomic object*
  2. Determine recursive step that determines how objects can be combined to produce another object closer to the atomic object
  3. Every recursive step must make progress towards base case: cannot use circular definitions or we get infinite recursion
- **Why use recursion?** Simplifies producing code for repetitive operations
- **Why not use recursion?** Uses more memory, unwinding can be slow – in some cases may re-calculate known values.

BST Implementation

Linked Nodes

- Data field
- Pointer to left, right child
- Pointer to parent (optional)

Pros

- Dynamic memory sizing

Cons

- $O(\log_2 n)$ access, $O(n^+)$ storage → worse than array
- Recursive read/update functions more complicated than array indexing

Recursion and Trees

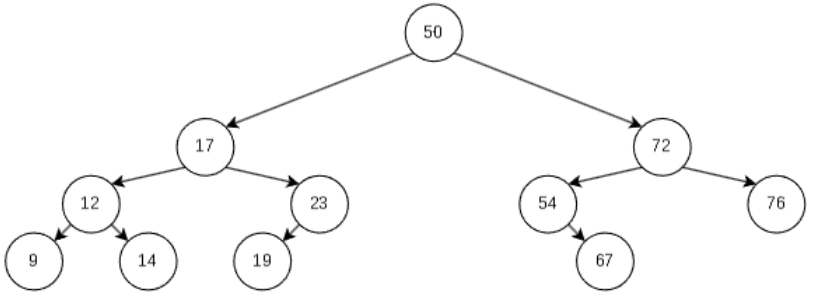
Property, access and update functions use recursion. Examples:

- $size(tree) = size(\text{left subtree}) + size(\text{right subtree}) + 1$ → Base case: empty subtree has size 0
- $height(tree) = max(height(\text{left subtree}), height(\text{right subtree}) + 1$ → Base case: empty subtree has size 0
- $min(tree) = min(\text{left subtree})$ → Base case: min value of subtree of a leaf node is leaf node's value.
- Insert, deletion, search, traversal all easiest to implement using recursive operations on subtrees

BST Implementation

Comparing Array and Linked Node Storage

Both trees have same contents and structure:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
50	17	72	12	23	54	76	9	14	19			67			

Search

- Almost identical to insert, except we return reference to found node.
- Use recursion to check if value is greater, lesser, or equal to the current node
  - If lesser: go to left subtree, check again
  - If greater: go to right subtree, check again
  - If equal: return reference
  - If null: value not found, return nullptr

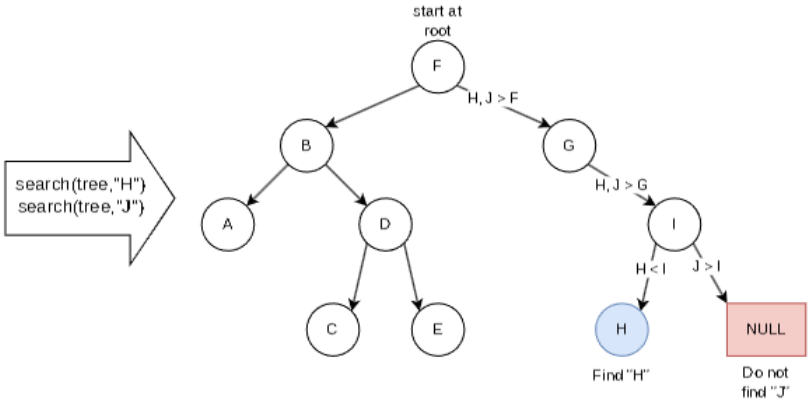
```
recursiveFind(node, data)
  if(data == node.data)
    return node.data
  else if (data < node.data)
    if(node.left == NULL)
      return NULL;
    else
      return read(node.left, data)
  else if (data > node.data)
    if(node.right == NULL)
      return NULL;
    else
      return read(node.right, data)
```


Insertion

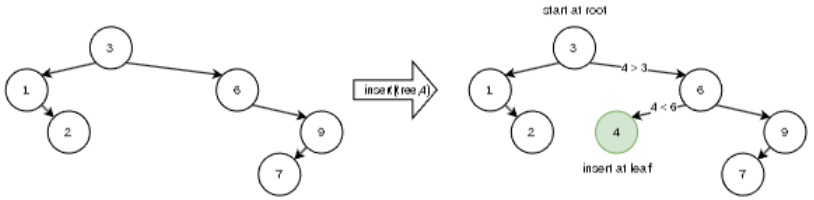
- Always start recursion at root, insert at leaf node.
- If tree is empty, inserted value becomes root node.
- Use recursion to check if value is greater, lesser, or equal to the current node
  - If lesser: go to left subtree, check again
  - If greater: go to right subtree, check again
  - If equal: value is already in tree, no duplicates allowed

```
recursiveInsert(node, data)
  if (data < node.data)
    if(node.left == NULL)
      addLeaf(node, data)
    else
      insert(node.left, data)
  else if (data > node.data)
    if(node.right == NULL)
      addLeaf(node, data);
    else
      insert(node.right, data)
  else
    pass
```


Insertion



Insertion



```
1  removeLeaf(node)
2    // disconnect parents or empty tree
3    if (node is right child)
4      node->parent->right = NULL
5    else if (node is left child)
6      node->parent->left = NULL
7    else (node is root)
8      tree->root = NULL
9
10   // delete the node
11   node->left,right,parent = NULL
12   return node
```

- Easiest case: disconnect from parent, return pointer to the deleted leaf.
- What if the leaf is also the tree's root? Set tree's root to NULL signifying an empty tree.

Basic deletion process:

1. Find the node to be removed by its value
2. Unlink the node from the tree
3. Re-link the removed node's parent/children if necessary
4. Return a reference to the removed node and/or free the node as required

Every deletion is one of three cases:

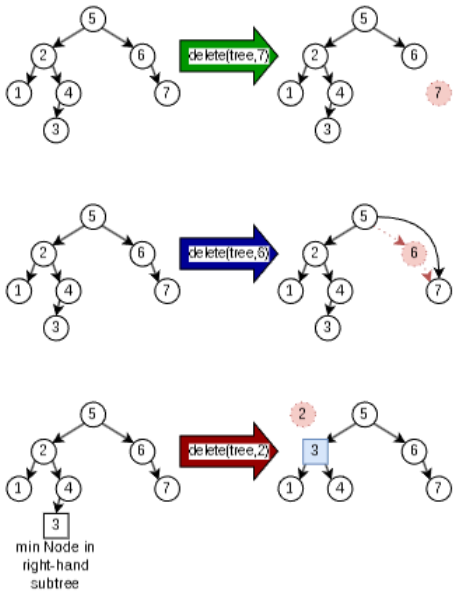
- Delete a leaf node → easiest case
- Delete a interior node with order 1 → can't just delete node because then the tree becomes un-connected.
- Delete an interior node with order 2 → must choose which child becomes the new parent


```
shortCircuit(node)
  if (node is right child)
    if (node has right child)
      node->parent->right = node->right
    else (node has left child)
      node->parent->right = node->left
  else if (node is left child)
    if (node has right child)
      node->parent->left = node->right
    else (node has left child)
      node->parent->left = node->left

  // delete the node
  node->left,right,parent = NULL
  return node
```

- Sets the child node's child to be the child of the parent, then deletes the node
- What if the node to be deleted is the root? Promote the child to new root
- When deleting a parent node in BST, ensure that the BST ordering property is preserved!


```
node = RecursiveFind(tree->root, data)
if (node not in BST)
    return NULL
else if (node has 0 subtrees)
    return deleteLeaf(node)
else if (node has 1 subtree)
    return shortCircuit(node)
else if (node has 2 subtrees)
    return promotion(node)
```



Traversal

Traversal

A specialized form of a graph *walk* that begins at a tree's root node, and visits each node exactly once. Traversals are categorized by the order in which the nodes are visited.

Two types of traversal:

Depth-first traversal The search tree is deepened as far as possible before visiting a sibling node. Deferred visits are stored in a LIFO data structure such as a stack.

Breadth-first traversal The search tree is broadened as far as possible before visiting a child level. Deferred visits are stored in a FIFO data structure such as a queue.

Deletion

promotion()

- We must promote a node to a higher space in the tree to preserve BST connectedness property!
- There are two possible "successor" candidates, it doesn't matter which is picked, both preserve BST ordering property:
  - `max` node in the left subtree → Traverse left once, then right as far as possible
  - `min` node in the right subtree → Traverse right once, then left as far as possible
- Successor candidate may be more than one level below the deletion candidate!
- **What if subtree `min` or `max` is not a leaf node?** Use short circuit to remove it and promote its child in place.
- **What if deletion candidate is also the tree's root node?** Successor candidate must also become tree's new root node.

Depth First Traversal

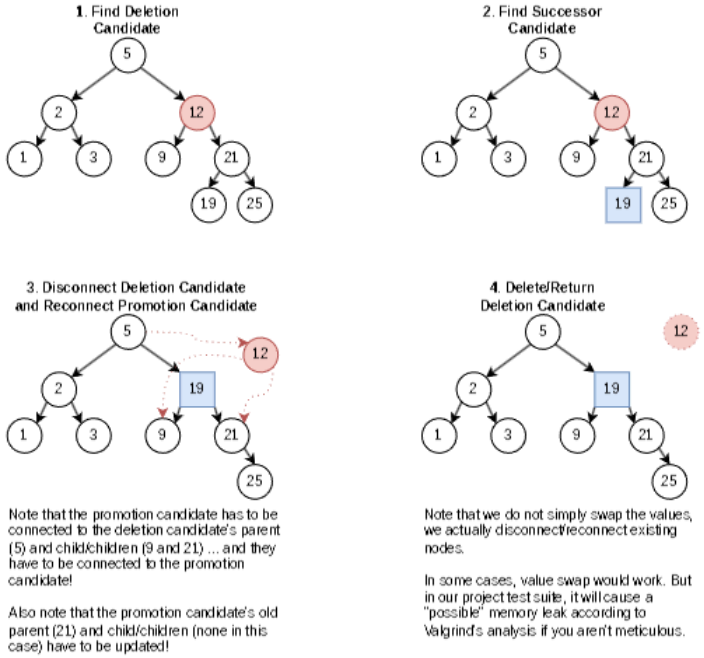
Basic Algorithm:

1. Start at root node
2. Recursively perform the following operations:
   1. If the current node is empty, then return.
   2. Otherwise, perform the following operations in a certain order:
      - L: Recursively traverse the current node's left subtree
      - N: Visit the current node
      - R: Recursively traverse the current node's right subtree

Visit order is described as **Pre-order** (NLR/NRL), **In-order** (LNR/RNL), or **Post-order** (LRN/RLN)

Deletion

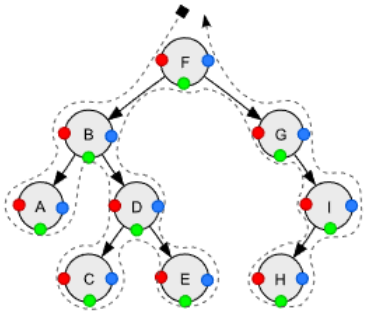
promotion()



Depth-first Traversal

Post-order (LRN)

```
1 RecursivePostorder(node):
2 RecursivePostorder(node->left)
3 RecursivePostorder(node->right)
4 Visit(node)
```

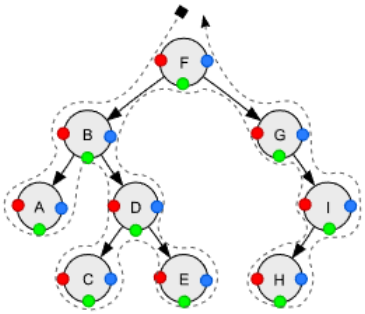


Following blue dots produces A, C, E, D, B, H, I, G, F → generates *reverse topological sort* and used to delete the tree

Depth-first Traversal

Pre-order (NLR)

```
1 RecursivePreorder(node):
2 Visit(node)
3 RecursivePreorder(node->left)
4 RecursivePreorder(node->right)
```

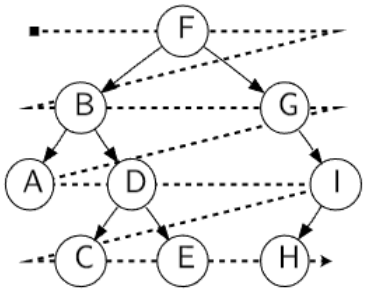


Following red dots produces F, B, A, D, C, E, G, I, H → *topological sort* used for copying the tree

Breadth-first Traversal

Level-order

```
1 RecursiveBFS(tree):
2 Queue q = empty
3 Enqueue(q, tree->root)
4 while q is not empty:
5 node = Dequeue(q)
6 Visit(node)
7 for child in node->children:
8 Enqueue(q, child)
```

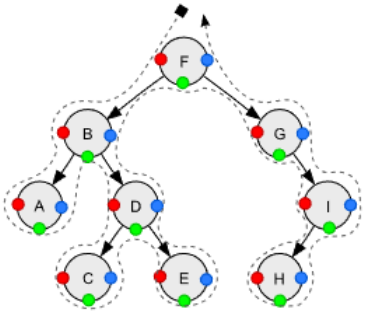


Nodes are visited in order of addition to the BFS queue. Produces F, B, G, A, D, I, C, E, H

Depth-first Traversal

In-order (LNR)

```
1 RecursiveInorder(node):
2 RecursiveInorder(node->left)
3 Visit(node)
4 RecursiveInorder(node->right)
```



Following green dots produces A, B, C, D, E, F, G, H, I → *lexicographical sort* used to give nodes in order of tree's underlying comparison function and to balance the tree

Array Based Heaps

Recall that we can map elements of a tree into a zero-indexed array:

function LEFT(i) ▷ The left child of the parent at i
  return $2i + 1$
end function

function RIGHT(i) ▷ The right child of the parent at i
  return $2i + 2$
end function

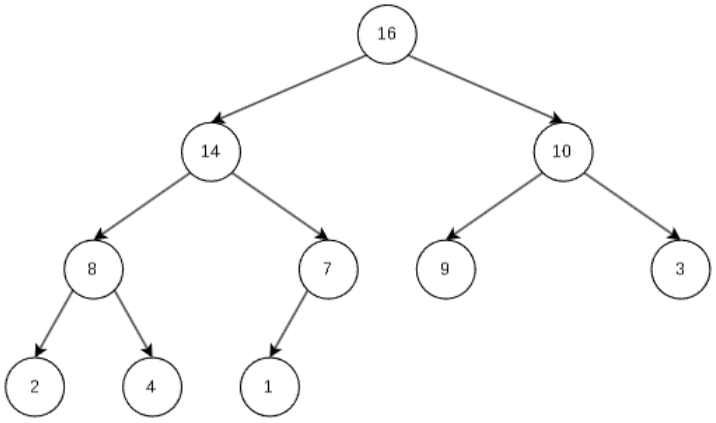
function PARENT(i) ▷ The parent of the child at index i
  return $\lfloor \frac{i-1}{2} \rfloor$
end function

- The priority member is always at $A[0]$
- There will never be an unfilled node at a smaller index than a filled node's index because the heap is *complete*.

Heaps

Array Based Heaps

Heap A														
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	16	14	10	8	7	9	3	2	4	1					



Heap Definition

Heap

A partially-ordered, complete tree that satisfies the *heap property*.

Partially Ordered Ancestors and descendents have an ordered relationship according to the *heap property*.

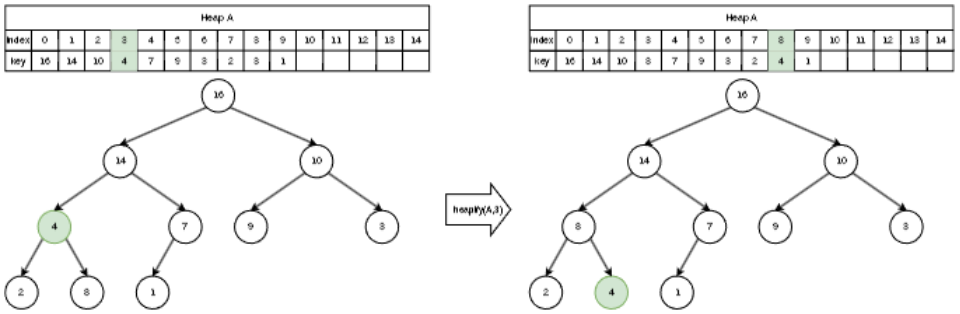
- The heap is not sorted. There is no ordered traversal of the nodes.
- There is no ordering between two sibling or cousin nodes like in a BST.

Complete All levels are full, except for possibly the last level, and the nodes in that level are as far left as possible.

Heap Property For any given node C and its' parent node P, then

- In a *max heap*, $key(P) \geq key(C)$.
- In a *min heap*, the $key(P) \leq key(C)$.

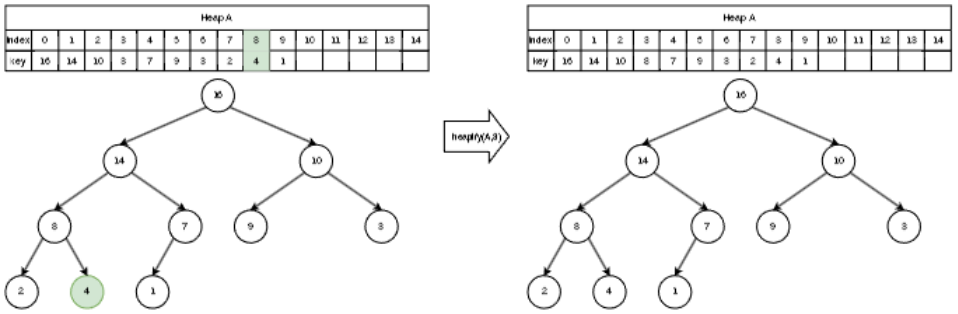
Maintaining the Heap Property



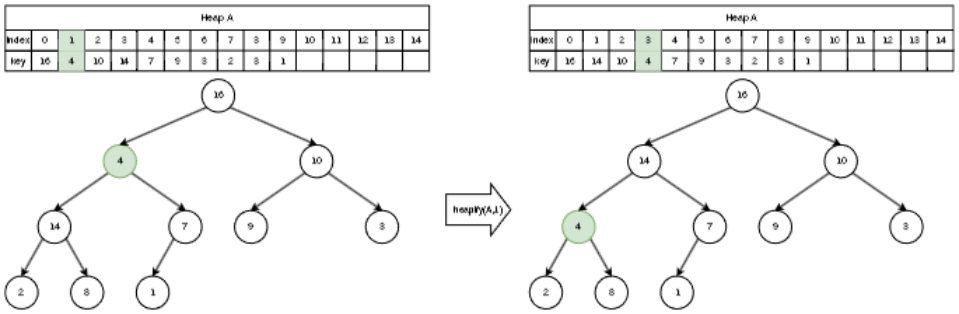
Maintaining the Heap Property

```
1: function HEAPIFY(A, i)
2:     l ← LEFT(i)
3:     r ← RIGHT(i)
4:     if l < size[A] and A[l] > A[i] then
5:         largest ← l ▷ largest is index of max key in parent-children subheap
6:     else
7:         largest ← i
8:     end if
9:     if r < size[A] and A[r] > A[i] then
10:        largest ← r
11:    end if
12:    if largest ≠ i then
13:        exchange A[i] ↔ A[largest]                    ▷ "Sift Down"
14:        HEAPIFY(A, largest)
15:    end if
16: end function
```

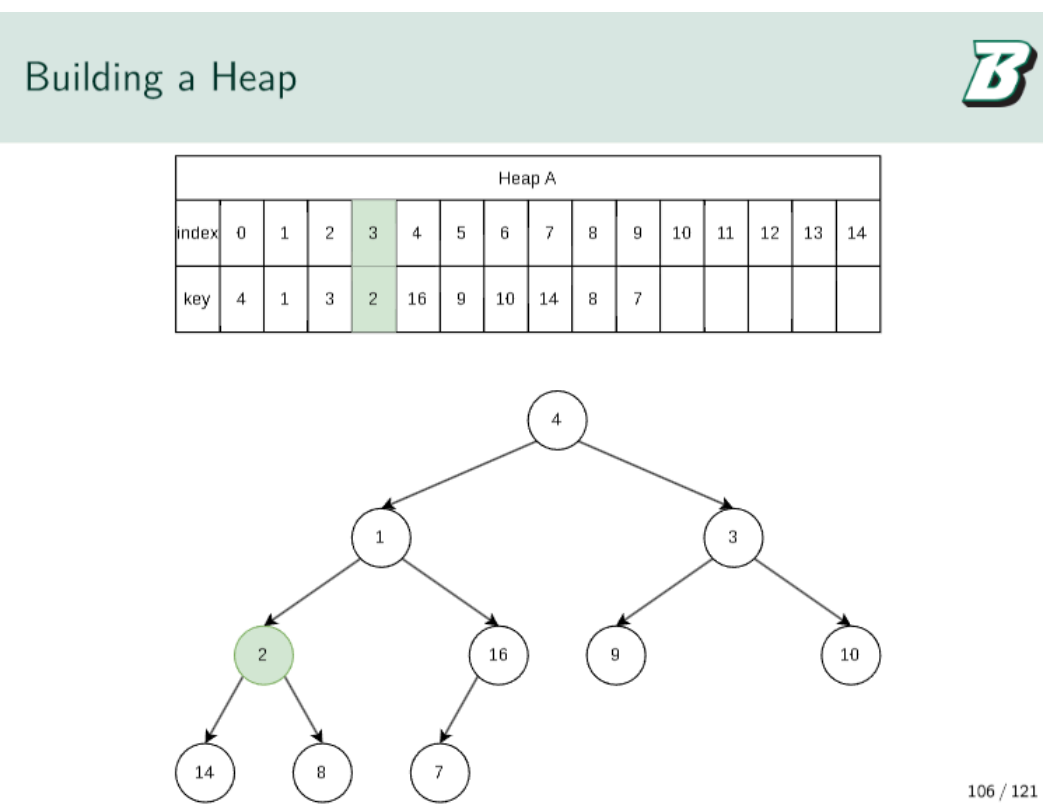

Maintaining the Heap Property



Maintaining the Heap Property



Building a Heap



Heap A															
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	4	1	3	2	16	9	10	14	8	7					

Building a Heap

```
1: function BUILD-HEAP(A)
2:     heapsize[A] ← length[A]          ▷ Convert array to heap object
3:     n ← ⌊(length[A]/2) − 1⌋          ▷ index of last interior node
4:     for i ← n downto 0 do
5:         HEAPIFY(A, i)                ▷ "Sift Down"
6:     end for
7: end function
```


Building a Heap

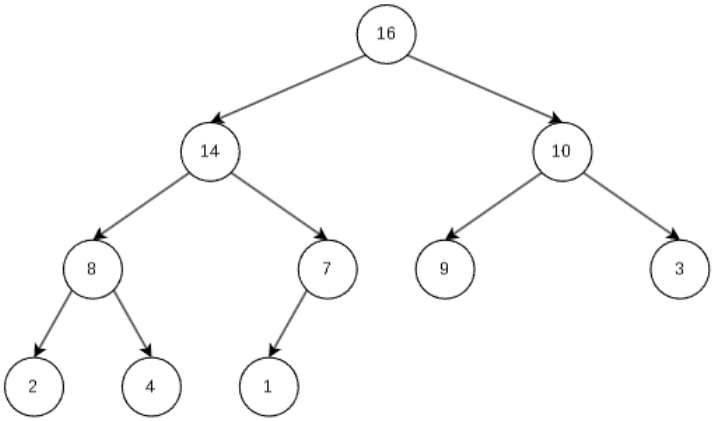
Heap A															
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	4	1	3	14	16	9	10	2	8	7					

Building a Heap

Heap A															
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	4	1	3	14	16	9	10	2	8	7					

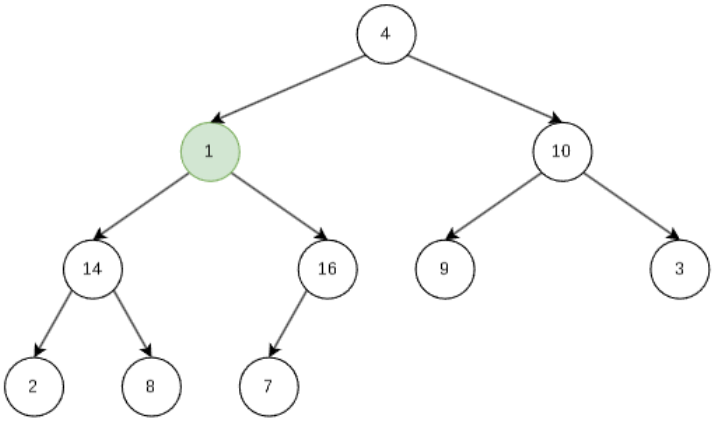
Building a Heap

Heap A															
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	16	14	10	8	7	9	3	2	4	1					



Building a Heap

Heap A															
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	4	1	10	14	16	9	3	2	8	7					

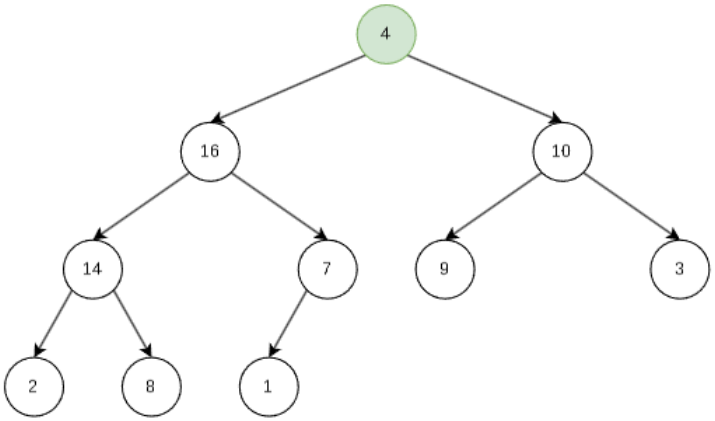


Sorting an Array using Heaps

```
1: function HEAPSORT(A)
2:     length ← heapsize[A]
3:     for i ← length downto 1 do
4:         exchange A[0] ↔ A[i]                    ▷ "Sift Up"
5:         heapsize[A] ← heapsize[A] − 1
6:         HEAPIFY(A, 0)
7:     end for
8:     heapsize[A] ← length
9:     return A  ▷ array in reverse priority order, no longer a heap
10: end function
```

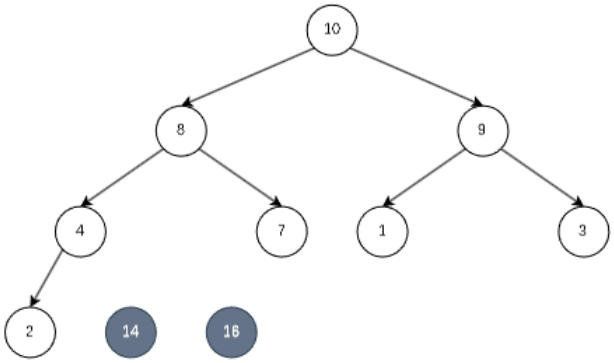

Building a Heap

Heap A															
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	4	16	10	14	7	9	3	2	8	1					



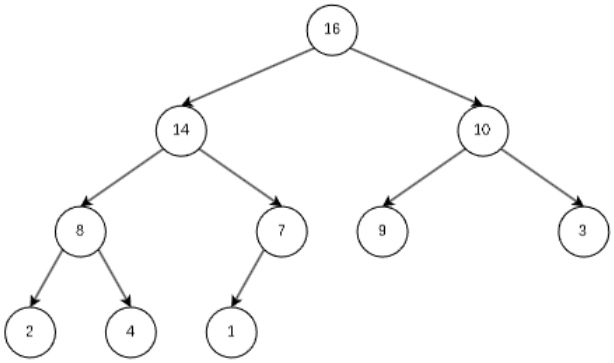
Sorting an Array using Heaps

Heap A														
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	10	8	9	4	7	1	3	2	14	16					



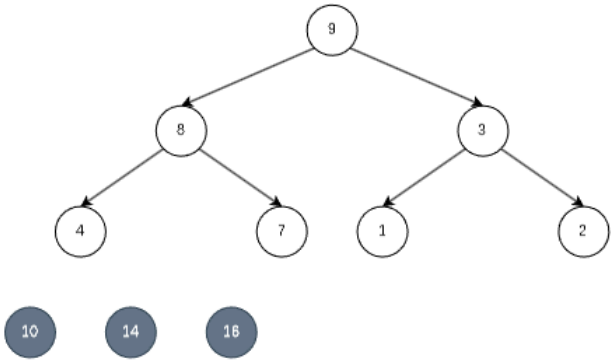
Sorting an Array using Heaps

Heap A														
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	16	14	10	8	7	9	3	2	4	1					



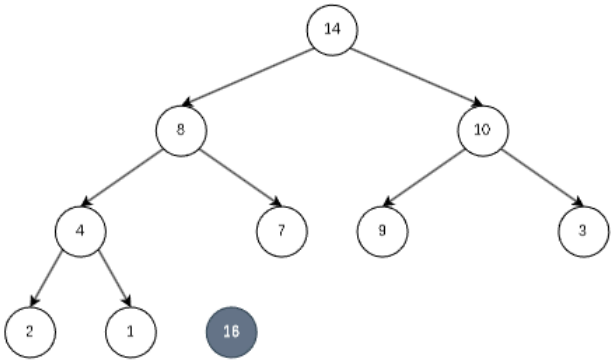
Sorting an Array using Heaps

Heap A														
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	16	14	10	8	7	9	3	10	14	16					

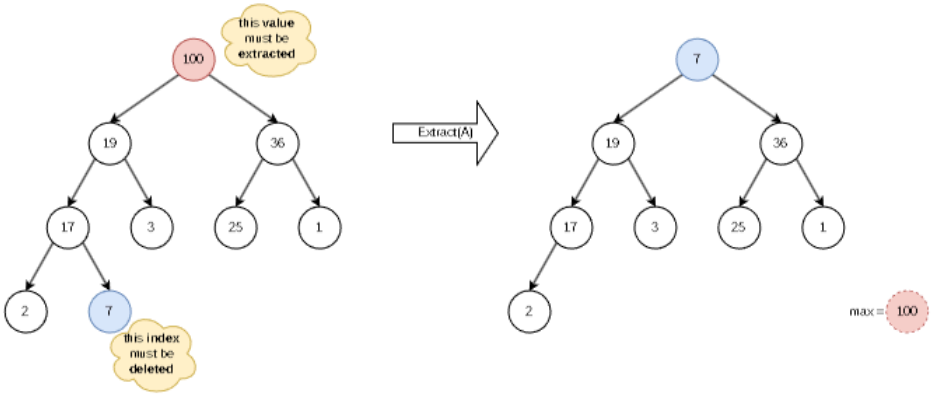


Sorting an Array using Heaps

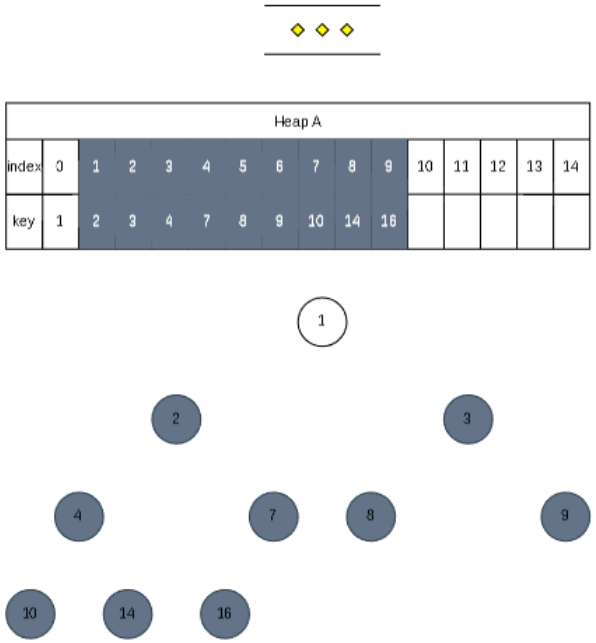
Heap A														
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	14	8	10	4	7	9	3	2	1	16					



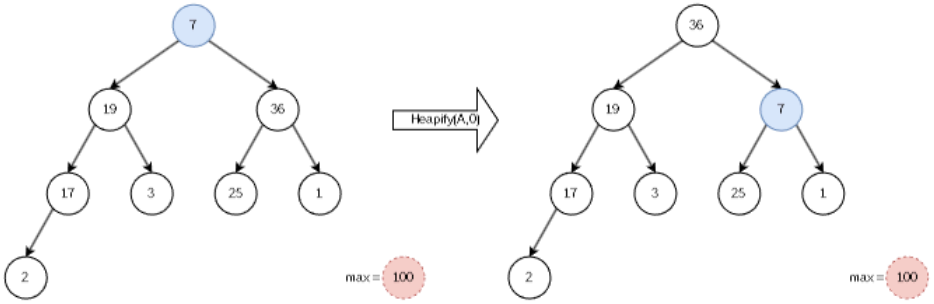
Getting the "Best" Element



Sorting an Array using Heaps



Getting the "Best" Element

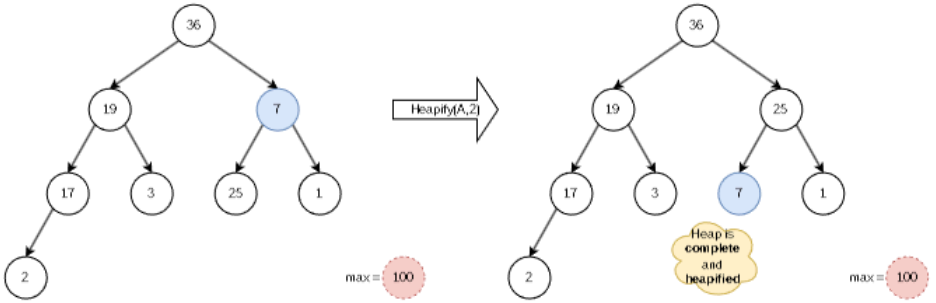


Getting the "Best" Element

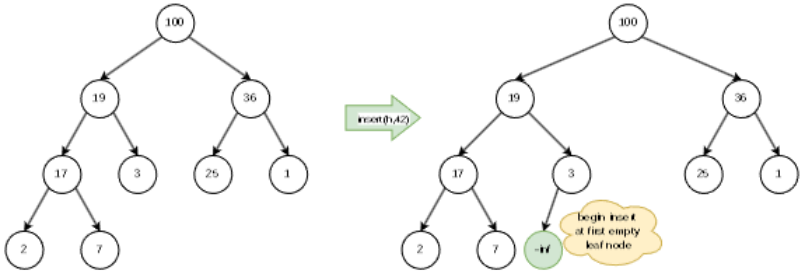
1: **function** MAXIMUM(A)
2: **return** $A[0]$
3: **end function**

1: **function** EXTRACT(A)
2: **if** $heapsize[A] < 1$ **then**
3: **error** "heap underflow"
4: **end if**
5: $max \leftarrow A[0]$
6: $A[0] \leftarrow A[heapsize[A] - 1]$
7: $heapsize[A] \leftarrow heapsize[A] - 1$
8: HEAPIFY($A, 0$) ▷ "Sift Down"
9: **return** max
10: **end function**

Getting the "Best" Element



Updating and Inserting Keys

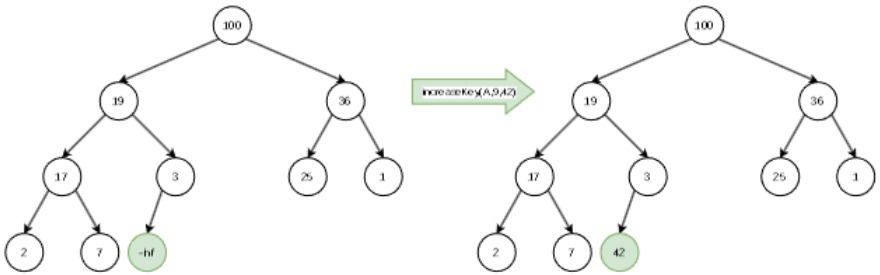


Updating and Inserting Keys

1: **function** IncreaseKey(A, i, key)
2: **if** $key < A[i]$ **then**
3: **error** "new key is smaller than old key"
4: **end if**
5: $A[i] \leftarrow key$
6: **while** $i > 0$ and $A[\text{Parent}(i)] < A[i]$ **do**
7: exchange $A[i] \leftrightarrow A[\text{Parent}(i)]$ ▷ "Sift Up"
8: $i \leftarrow \text{Parent}(i)$
9: **end while**
10: **end function**

1: **function** Insert(A, key)
2: $heapsize[A] \leftarrow heapsize[A] + 1$
3: $A[heapsize[A] - 1] \leftarrow -\infty$
4: IncreaseKey($A, heapsize[A] - 1, key$)
5: **end function**

Updating and Inserting Keys



Priority Queue

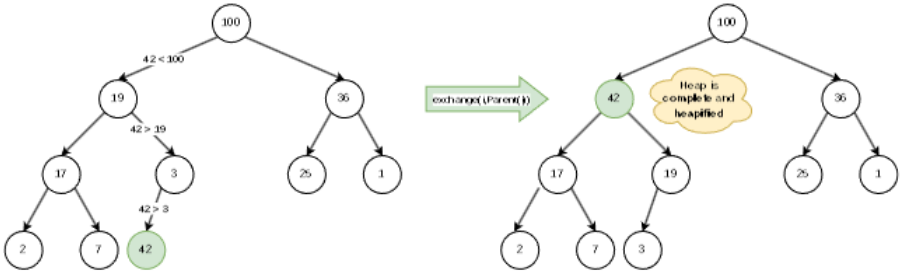
A specialized implementation of queues, priority queues are queues where insertion location is based on a *priority value*. Priority queues are a sequential collection:

Sequential Priority of elements causes ordering

Collection Contains items of the same type, forming a grouping

- Priority queues are an ADT providing restricted access control through encapsulation
- Like queues, only one element can be removed (queue: first in, priority queue: highest priority).
- Unlike queues, insertion happens anywhere in the priority queue, not just the tail.
- Don't confuse an item's *priority* with its *value*
  - Consider a priority queue in a scheduler:
  - → value is the process' PID
  - → priority is the preference to schedule that process (least recently run, lowest "niceness", etc.)

Updating and Inserting Keys

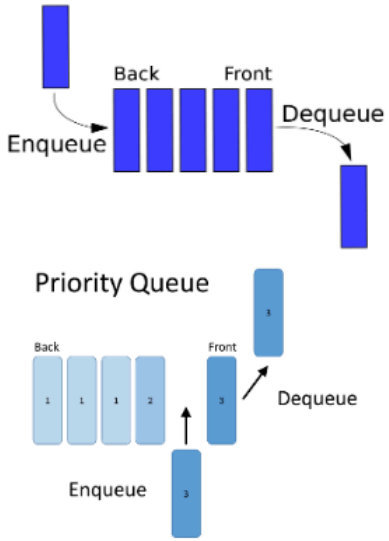


Priority Queue Access Control

- Queues are **First In, First Out** (FIFO) ...
- But priority queues are **"Best" Priority Served First**
- Priority queues implement operations congruent to sorting algorithms
- Unlike queues, priority queues are often not built with vectors, but heaps or binary search trees!
- Think of priority queues being like airport queues where pilots and flight crew

Priority Queues

Queue vs Priority Queue



Examples of Priority Queues

What are some examples of priority queues in computer science?

Design Considerations

Choosing an underlying ADT implementation can have a dramatic effect on the performance of a priority queue.

Array
- Sorted array makes insertion slow, but removal fast
- Unsorted array makes insertion fast, but removal slow

Linked List
- Sorted list makes insertion slow, but removal fast
- Unsorted list makes insertion fast, but removal slow

Binary Search Tree
- Insertion and removal are both fast as long as the tree is balanced
- Duplicate keys not possible – useful for set-like queues which enforce uniqueness.

Examples of Priority Queues

What are some examples of priority queues in computer science?

- Priority scheduling (as implemented in early Linux and XV6)
- Bandwith management in network routing
- Huffman coding for data compression
- Efficient path finding algorithms (A* algorithm)
- Efficient spanning algorithms (Prim's algorithm)

Thought questions:

- Does it matter if the Priority Queue's storage list is singly or doubly linked?
- What inputs would cause a Binary Search Tree to be unbalanced?
- If the tree isn't balanced, why is performance is basically equivalent to a sorted linked list?

- Insertion and removal of the priority item are both logarithmic time.
- Inspection (without removal) of the priority item is constant time.
- Search for a particular key is linear time and generally not supported.
- Duplicate keys permitted, but no way to resolve "ties" in priority without refactoring our removal method.

Thought questions:

- If the priority item is always at the root, why isn't removing it constant time?
- Why can't we search for a particular item quickly if the heap property is conserved?
- Why are duplicates a problem for heap-based priority queues?
- What changes would we have to make to change from a max-priority queue to a min-priority queue?