66

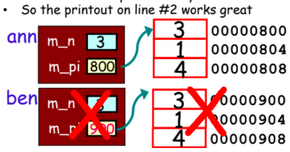
The Copy Constructor

```
class PiNerd
{
public:
  PiNerd(int n) { … }
  ~PiNerd(){ delete[]m_pi; }

  // copy constructor
  PiNerd(const PiNerd &src)
  {
    m_n = src.m_n;
    m_pi = new int[m_n];
    for (int j=0;j<m_n;j++)
      m_pi[j] = src.m_pi[j];
  }

  void showOff() { ... }

private:
  int *m_pi, m_n;
};
```

```
int main()
{
  PiNerd ann(3);
  if (...)
  {
     PiNerd ben = ann;
     ...
  } // ben's d'tor called #1

  ann.showOff(); // #2
}
```

3
1
4

- When we hit line #1, ben's destructor runs and deletes ben's array at location 900
- At the end of the destructor, the ben variable disappears too
- But notice that ann is just fine, since her variable has a separate array at 800
- So the printout on line #2 works great



We're A-OK, since ann still has its own array!

14

The Assignment Operator

The fix:

Our assignment operator function must check to see if a variable is being assigned to itself, and if so, do nothing…

```
class PiNerd
{
  ...
  PiNerd &operator=(const PiNerd &src)
  {
    if (&src == this)  // #1
      return *this; // do nothing
    delete [] m_pi;
    m_n = src.m_n;
    m_pi = new int[m_n];
    for (int j=0;j<m_n;j++)
       m_pi[j] = src.m_pi[j];
    return *this;
  }
  …
};
```

- The solution is to check to see if the parameter (src) has the same address as the target object that's supposed to be changed.
- We can get the target object's address with the this keyword.
- If the target object has the same address as its parameter (line #1) then we're assigning an object to itself (probably via an alias).
- In this case, we don't need to do any assignment.
- Instead, we just return a reference to the target object (or to src – they're the same!) and exit our function.

54

Linked List Cheat Sheet

```
struct Node
{
   string value;

   Node *next;
   Node  *prev;
};
```

Given a pointer to a node: Node *ptr;

NEVER access a node's data until validating its pointer:
```
if (ptr != nullptr)
    cout << ptr->value;
```

To advance ptr to the next node/end of the list:
```
if (ptr != nullptr)
    ptr = ptr->next;
```

To see if ptr points to the last node in a list:
```
if (ptr != nullptr && ptr->next == nullptr)
    then-ptr-points-to-last-node;
```

To get to the next node's data:
```
if (ptr != nullptr && ptr->next != nullptr)
    cout << ptr->next->value;
```

To get the head node's data:
```
if (head != nullptr)
    cout << head->value;
```

To check if a list is empty:
```
if (head == nullptr)
    cout << "List is empty";
```

Does our traversal meet this requirement?

```
NODE *ptr = head;
while (ptr != nullptr)
{
   cout << ptr->value;
   ptr = ptr->next;
}
```

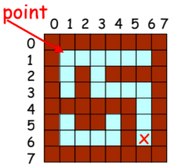
To check if a pointer points to the first node in a list:
```
if (ptr == head)
   cout << "ptr is first node";
```

26

Solving a Maze with a Stack!

1. PUSH starting point onto the stack.
2. Mark the starting point as "discovered."
3. While the stack is not empty:
   A. POP the top point off the stack into a variable.
   B. If we're at the endpoint, DONE! Otherwise…
   C. If slot to the WEST is open & is undiscovered
      Mark (curx-1,cury) as "discovered"
      PUSH (curx-1,cury) on stack.
   D. If slot to the EAST is open & is undiscovered
      Mark (curx+1,cury) as "discovered"
      PUSH (curx+1,cury) on stack.
   E. If slot to the NORTH is open & is undiscovered
      Mark (curx,cury-1) as "discovered"
      PUSH (curx,cury-1) on stack.
   F. If slot to the SOUTH is open & is undiscovered
      Mark (curx,cury+1) as "discovered"
      PUSH (curx,cury+1) on stack.
4. If the stack is empty and we haven't reached our goal position, then the maze is unsolvable.

starting point
x=1,y=1



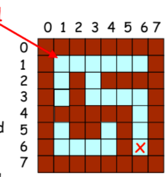
The stack-based maze-solving algorithm is called a "depth-first search" because the use of a stack causes it to explore "deep" down passages 'til it hits a dead end, then unravel back to the last junction where it again explores deep to another dead end, etc. **It's very similar to the recursive maze searching algorithm.**

35

Solving a Maze with a Queue!

(AKA Breadth-first Search)

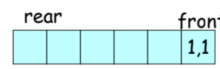
sx,sy = 1,1



1. Insert starting point onto the queue.
2. Mark the starting point as "discovered."
3. While the queue is not empty:
   A. Remove the front point from the queue.
   B. If we're at the endpoint, DONE! Otherwise…
   C. If slot to the WEST is open & is undiscovered
      Mark (curx-1,cury) as "discovered"
      INSERT (curx-1,cury) on queue.
   D. If slot to the EAST is open & is undiscovered
      Mark (curx+1,cury) as "discovered"
      INSERT (curx+1,cury) on queue.
   E. If slot to the NORTH is open & is undiscovered
      Mark (curx,cury-1) as "discovered"
      INSERT (curx,cury-1) on queue.
   F. If slot to the SOUTH is open & is undiscovered
      Mark (curx,cury+1) as "discovered"
      INSERT (curx,cury+1) on queue.
4. If the queue is empty and we haven't reached our goal position, then the maze is unsolvable.

And so on...

curx,cury=



6

Inheritance

Review

Inheritance is a way to form new classes using classes that have already been defined.

Reuse

Reuse is when you write code once in a base class and reuse the same code in your derived classes (to save time).

Extension

Extension is when you add new behaviors (member functions) or data to a derived class that were not present in a base class.

Car → void accelerate(), void brake(), void turn(float angle)
Bat Mobile: public Car → void shootLaser(float angle)

Specialization

Specialization is when you redefine an existing behavior (from the base class) with a new behavior (in your derived class).

Car → void accelerate() { addSpeed(10); }
Bat Mobile: public Car → void accelerate() { addSpeed(200); }

46

Polymorphism Cheat Sheet

You can't access private members of the base class from the derived class:

```
// BAD!
class Base
{
public:
…



private:
   int v;
};

class Derived: public Base
{
public:

   Derived(int q)
   {
      v = q;  // ERROR!
   }


   void foo()
   {
     v = 10; // ERROR!
   }
};
```

```
// GOOD!
class Base
{
public:
   Base(int x)
     { v = x; }
   void setV(int x)
     { v = x; }
   …
private:
   int v;
};

class Derived: public Base
{
public:

   Derived(int q)
    : Base(q)   // GOOD!
   {
     …
   }

   void foo()
   {
     setV(10);  // GOOD!
   }
};
```

Always make sure to add a virtual destructor to your base class:

```
// BAD!
class Base
{
public:
  ~Base() { … } // BAD!
  …
};

class Derived: public Base
{
  …
};
```

```
// GOOD!
class Base
{
public:
 virtual ~Base() { … } // GOOD!
  …
};

class Derived: public Base
{
  …
};
```

```
class Person
{
public:
  virtual void talk(string &s) { … }
  …
};

class Professor: public Person
{
public:
    void talk(std::string &s)
    {
       cout << "I profess the following: ";
       Person::talk(s); // uses Person's talk
    }
};
```

Don't forget to use virtual to define methods in your base class, if you expect to re-define them in your derived class(es)

To call a base-class method that has been re-defined in a derived class, use the base:: prefix!

So long as you define your BASE version of a function with virtual, all derived versions of the function will automatically be virtual too (even without the virtual keyword)!

47

Polymorphism Cheat Sheet, Page #2

```
class SomeBaseClass
{
public:
   virtual void aVirtualFunc() { cout << "I'm virtual"; } // #1
   void notVirtualFunc() { cout << "I'm not"; }           // #2
   void tricky()                                          // #3
   {
      aVirtualFunc();                                     // ***
      notVirtualFunc();
   }
};

class SomeDerivedClass: public SomeBaseClass
{
public:
   void aVirtualFunc()   { cout << "Also virtual!"; }     // #4
   void notVirtualFunc() { cout << "Still not"; }         // #5
};

int main()
{
   SomeDerivedClass d;
   SomeBaseClass  *b = &d;  // base ptr points to derived obj

   // Example #1
   cout << b->aVirtualFunc();      // calls function #4

   // Example #2
   cout << b->notVirtualFunc(); // calls function #2

   // Example #3
   b->tricky();       // calls func #3 which calls #4 then #2
}
```

Example #1: When you use a BASE pointer to access a DERIVED object, AND you call a VIRTUAL function defined in both the BASE and the DERIVED classes, your code will call the DERIVED version of the function.

Example #2: When you use a BASE pointer to access a DERIVED object, AND you call a NON-VIRTUAL function defined in both the BASE and the DERIVED classes, your code will call the BASE version of the function.

Example #3: When you use a BASE pointer to access a DERIVED object, all function calls to VIRTUAL functions (***) will be directed to the derived object's version, even if the function (tricky) calling the virtual function is NOT VIRTUAL itself.

45

Writing Recursive Functions: A Critical Tip!

Your recursive function should generally only access the current node/array cell passed into it!

Your recursive function should rarely/never access the value(s) in the node(s)/cell(s) below it!

```
// good examples!
int recursiveGood(Node *p)
{
  …
  if (p->value == someValue)
    do something;

  if (p == nullptr || p->next == nullptr)
    do something;

  int v = p->value +
       recursiveGood(p->next);

  if (p->value > recursiveGood(p->next))
    do something;
}
```

```
// bad examples!!!
int recursiveBad(Node *p)
{
  …
  if (p->next->value == someValue)
    do something;

  if (p->next->next == nullptr)
    do something;

  int v = p->value + p->next->value +
       recursiveBad(p->next->next);

  if (p->value > p->next->value)
    do something;
}
```

46

Writing Recursive Functions: A Critical Tip!

Your recursive function should generally only access the current node/array cell passed into it!

Your recursive function should rarely/never access the value(s) in the node(s)/cell(s) below it!

```
// good examples!
int recursiveGood(int a[], int count)
{
  …
  if (count == 0 || count == 1)
    do something;

  if (a[0] == someValue)
    do something;

  int v = a[0] +
       recursiveGood(a+1, count-1);

  if (a[0] > recursiveGood(a+1, count-1))
    do something;
}
```

```
// bad examples!!!
int recursiveBad(int a[], int count)
{
  …
  if (count == 2)
    do something;

  if (a[1] == someValue)
    do something;

  int v = a[0] + a[1] +
       recursiveBad(a+2,count-2);

  if (a[0] > a[1])
    recursiveBad(a+2,count-2);
}
```

18

Carey's Template Cheat Sheet

- To templatize a non-class function called bar:
  - Update the function header: int bar(int a) → template <typename ItemType> ItemType bar(ItemType a);
  - Replace appropriate types in the function to the new ItemType: { int a; float b; … } → {ItemType a; float b; …}
- To templatize a class called foo:
  - Put this in front of the class declaration: class foo { … }; → template <typename ItemType> class foo { … };
  - Update appropriate types in the class to the new ItemType
  - How to update internally-defined methods:
    - For normal methods, just update all types to ItemType: int bar(int a) { … } → ItemType bar(ItemType a) { … }
    - Assignment operator: foo &operator=(const foo &other) → foo<ItemType>& operator=(const foo<ItemType>& other)
    - Copy constructor: foo(const foo &other) → foo(const foo<ItemType> &other)
  - For each externally defined method:
    - For non inline methods: int foo::bar(int a) → template <typename ItemType> ItemType foo<ItemType>::bar(ItemType a)
    - For inline methods: inline int foo::bar(int a) → template <typename ItemType> inline ItemType foo<ItemType>::bar(ItemType a)
    - For copy constructors and assignment operators
    - foo &foo::operator=(const foo &other) → foo<ItemType>& foo<ItemType>::operator=(const foo<ItemType>& other)
    - foo::foo(const foo &other) → foo<ItemType>::foo(const foo<ItemType> &other)
  - If you have an internally defined struct blah in a class: class foo { … struct blah { int val; }; … };
    - Simply replace appropriate internal variables in your struct (e.g., int val;) with your ItemType (e.g., ItemType val;)
  - If an internal method in a class is trying to return an internal struct (or a pointer to an internal struct):
    - You don't need to change the function's declaration at all inside the class declaration; just update variables to your ItemType
  - If an externally-defined method in a class is trying to return an internal struct (or a pointer to an internal struct):
    - Assuming your internal structure is called "blah", update your external function bar definitions as follows:
    - blah foo::bar(…) { … } → template<typename ItemType>typename foo<ItemType>::blah foo<ItemType>::bar(…) { … }
    - blah *foo::bar(…) { … } → template<typename ItemType>typename foo<ItemType>::blah *foo<ItemType>::bar(…) { … }
- Try to pass templated items by const reference if you can (to improve performance):
  - Bad: template <typename ItemType> void foo(ItemType x)
  - Good: template <typename ItemType> void foo(const ItemType &x)

53

* For more details, see: http://en.cppreference.com/w/cpp/container#Sequence_containers

Iterator Gotchas!

```
int main()
{
  vector<string>  x;

  x.push_back("Carey");
  x.push_back("Rick");
  x.push_back("Alex");

  vector<string>::iterator it;

  it = x.end();
  it--;    // it points at Alex

  x.push_back("Yong"); // add

  cout << *it;  // ERROR!
}
```

I'm no longer valid!!! ☹

Let's say you point an iterator to an item in a vector…

If you add an item anywhere to the vector you must assume your iterator is invalidated!

And if you erase that item or an item that comes before it, your iterator is also invalidated!

Why? When you add/erase items in a vector, it may shuffle its memory around (without telling you) and then your iterators may not point to the right place any more!

Leaving the old iterator pointing to a random spot in your PC's memory.

33

STL and Big Oh Cheat Sheet

When describing the Big-O of each operation (e.g. insert) on a container (e.g., a vector) below, we assume that the container holds n items when the operation is performed.

Name: list
Purpose: Linked list
Usage: list<int> x; x.push_back(5);
Inserting an item (top, middle*, or bottom): O(1)
Deleting an item (top, middle*, or bottom): O(1)
Accessing an item (top or bottom): O(1)
Accessing an item (middle): O(n)
Finding an item: O(n)
*But to get to the middle, you may have to first iterate through X items, at cost O(x)

Name: vector
Purpose: A resizable array
Usage: vector<int> v; v.push_back(42);
Inserting an item (top, or middle): O(n)
Inserting an item (bottom): O(1)
Deleting an item (top, or middle): O(n)
Deleting an item (bottom): O(1)
Accessing an item (top, middle, or bottom): O(1)
Finding an item: O(n)

Name: set
Purpose: Maintains a set of unique items
Usage: set<string> s; s.insert("Ack!");
Inserting a new item: $O(\log_2 n)$
Finding an item: $O(\log_2 n)$
Deleting an item: $O(\log_2 n)$

Name: map
Purpose: Maps one item to another
Usage: map<int,string> m; m[10] = "Bill";
Inserting a new item: $O(\log_2 n)$
Finding an item: $O(\log_2 n)$
Deleting an item: $O(\log_2 n)$

Name: queue and stack
Purpose: Classic stack/queue
Usage: queue<long> q; q.push(5);
Inserting a new item: O(1)
Popping an item: O(1)
Examining the top: O(1)

If instead of holding n items, a container holds p items, then just replace "n" with "p" when you do your analysis.

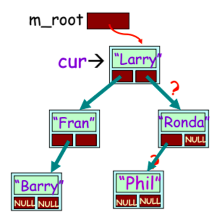
27

Sorting Overview

Sort Name	Stable/ Non-stable	Notes
Selection Sort	Unstable	Always $O(n^2)$, but simple to implement. Can be used with linked lists. Minimizes the number of item-swaps (important if swaps are slow)
Insertion Sort	Stable	O(n) for already or nearly-ordered arrays. $O(n^2)$ otherwise. Can be used with linked lists. Easy to implement.
Bubble Sort	Stable	O(n) for already or nearly-ordered arrays (with a good implementation). $O(n^2)$ otherwise. Can be used with linked lists. Easy to implement. Rarely a good answer on an interview!
Shell Sort	Unstable	$O(n^{1.25})$ approx. OK for linked lists. Used in some embedded systems (eg, in a car) instead of quicksort due to fixed RAM usage.
Quick Sort	Unstable	$O(n \log_2 n)$ average, $O(n^2)$ for already/mostly/reverse ordered arrays or arrays with the same value repeated many times. Can be used with linked lists. Can be parallelized across multiple cores. Can require up O(n) slots of extra RAM (for recursion) in the worst case, $O(\log_2 n)$ avg.
Merge Sort	Stable	$O(n \log_2 n)$ always. Used for sorting large amounts of data on disk (aka "external sorting"). Can be used to sort linked lists. Can be parallelized across multiple cores. Downside: Requires n slots of extra memory/disk for merging – other sorts don't need xtra RAM.
Heap Sort	Unstable	$O(n \log_2 n)$ always. Sometimes used in low-RAM embedded systems because of its performance/low memory req'ts.

34

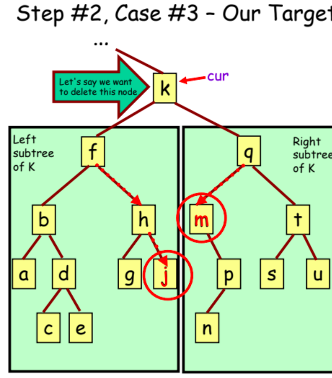
```
void insert(const std::string &value)
{
   if (m_root == NULL)
     {  m_root = new Node(value);   return; }

   Node *cur = m_root;
   for (;;)
   {
      if (value == cur->value)   return;
      if (value < cur->value)
      {
         if (cur->left != NULL)
            cur = cur->left;
         else
         {
            cur->left = new Node(value);
            return;
         }
      }
      else if (value > cur->value)
      {
         if (cur->right != NULL)
            cur = cur->right;
         else
         {
            cur->right = new Node(value);
            return;
         }
      }
   }
}
```




```
void main(void)
{
   BinarySearchTree bst;

   bst.insert("Larry");
   ...
   bst.insert("Phil");
}
```

14

Step #2, Case #3 – Our Target Node has Two Children



Well there's a trick! We don't actually delete the target node itself! Instead, we replace the value in the target node with a value from another node… and then we delete that other node instead!

Let's say our goal is to delete some target node k (see example to the left). Node k could be the root node, or some node in the middle of the tree. It doesn't matter.

The replacement value to be copied into node k MUST come from ONLY one of two places:

1. K's left subtree's largest-valued node
2. K's right subtree's smallest-valued node

To find the largest valued node in k's left subtree, we basically set a pointer to k's left child then keep following the right pointer of each node (zero or more times) until we can't go any further:

```
ptr = cur->left;
while (ptr->right != nullptr)
   ptr = ptr->right;
// ptr points at the largest valued node on the right
```

We could do use a similar algorithm with k's right child if we wanted to (going all the way to the left to find the smallest value in the right subtree). Either choice is valid, and you can pick one way to implement your code. There's no right or wrong way.

Once we find the new target node (either j or m in this case), then we just copy that node's value up into our root node and then delete that other target node instead!

w/ virtual

Construct:

Base Class Pet
private members
constructor

Derived class Pet
private mem
constructor Woof

Destruct:

Derived Class:
destructor Dog ran away
private mems ~Pet

Base Class:
destructor ~Pet
private mems

* think of it as a stack

54

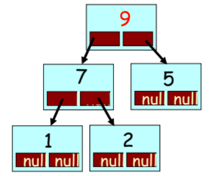
Hash Tables vs. Binary Search Trees

	Hash Tables	Binary Search Trees
Speed	O(1) regardless of # of items	$O(\log_2 N)$
Simplicity	Easy to implement	More complex to implement
Max Size	Closed: Limited by array size Open: Not limited, but high load impacts performance	Unlimited size
Space Efficiency	Wastes a lot of space if you have a large hash table holding few items	Only uses as much memory as needed (one node per item inserted)
Ordering	No ordering (random)	Alphabetical ordering

14

Extracting the Biggest Item

1. If the tree is empty, return error.
2. Otherwise, the top item in the tree is the biggest value. Remember it for later.
3. If the heap has only one node, then delete it and return the saved value.
4. Copy the value from the right-most node in the bottom-most row to the root node.
5. Delete the right-most node in the bottom-most row.
6. Repeatedly swap the just-moved value with the larger of its two children until the value is greater than or equal to both of its children. ("sifting DOWN")
7. Return the saved value to the user.



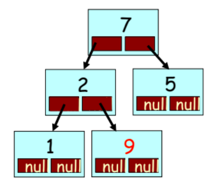
When we're done, the largest value is on the top again, and the heap is consistent.

16

Adding a Node to a Maxheap

(Let's see how to add a value of 9)

1. If the tree is empty, create a new root node & return.
2. Otherwise, insert the new node in the bottom-most, left-most position of the tree (so it's still a complete tree).
3. Compare the new value with its parent's value.
4. If the new value is greater than its parent's value, then swap them.
5. Repeat steps 3-4 until the new value rises to its proper place.



This process is called "reheapification."

63

The STL "find" Function

```
#include <list>
#include <algorithm>

int main()
{
  list<string>  names;
  ... // fill with a bunch of names

  list<string>::iterator a, b, itr;

  a = names.begin(); // start here
  b = names.end();   // end here

  itr = find( a , b , "Judy" );

  if (itr == b)
    cout << "I failed!";
  else
    cout << "Hello: " << *itr;
}
```

- The STL provides a find function that works with vectors/lists.
- (They don't have built-in find methods like map & set)
- Make sure to include the algorithm header file!
- The first argument is an iterator that points to where you want to start searching.
- The second argument is an iterator that points JUST AFTER where you want to stop searching!
- The final argument is what you're searching for.
- And just like set and map's find methods, this version returns an iterator to the item that it found.
- And if find couldn't locate the item, it will return whatever you passed in for the second parameter.
- So make sure to check for this value to see if the find function was successful!

- Virtual: always use for destructors & base
- Order of construction/destruction
  - Constructor: Base (priv members -> constructor) -> Derived (priv mem -> constr)
  - Destructor: Derived (destructor -> private mems) -> Base (destructor ->priv mems)
- Linked lists: Hash sets, Two pointers
- Post fix
  - Infix: A / (B * C) - (D / E ^ F) * G
  - Postfix: A B C * / D E F ^ / G * -
- Vector: Fast random access
- List: Lots of insert/erase not at end
- Recursion: Base case, Simplifying step
- Write func header, define magic func, add base case, solve problem w magic func, remove magic, validate func, write test cases

Sort Name	Stability	Notes
Selection Sort	Unstable	Always `O(n^2)`, but simple to implement. Can be used with linked lists. Minimizes the number of item-swaps (important if swaps are slow). ➜ find minimum then place at front, then continue looking at smaller array
Insertion Sort	Stable	`O(n)` for already or nearly-ordered arrays. `O(n^2)` otherwise. Can be used with linked lists. Easy to implement. ➜ sorts first two, then index 1 and 2, … so that minimum are sorted on the left
Bubble Sort	Stable	`O(n)` for already or nearly-ordered arrays (with a good implementation). `O(n^2)` otherwise. Can be used with linked lists. Easy to implement. Rarely a good answer on an interview! ➜ bubble up the maximum
Shell Sort	Unstable	Approximately `O(n^1.25)`. OK for linked lists. Used in some embedded systems (eg, in a car) instead of quicksort due to fixed RAM usage. ➜ breaks into intervals

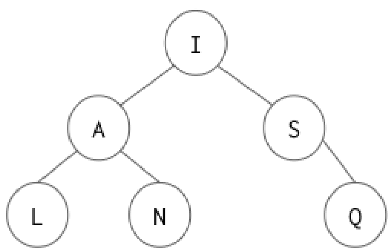
Sort Name	Stability	Notes
Quicksort	Unstable	`O(nlogn)` average, `O(n^2)` for already/mostly/reverse ordered arrays or arrays with the same value repeated many times. Can be used with linked lists. Can be parallelized across multiple cores. Can require up `O(n)` space (for recursion) in the worst case, `O(logn)` space average. ➜ pivot is sorted, partition all items before and <, all items after are >
Mergesort	Stable	`O(nlogn)` always. Used for sorting large amounts of data on disk (aka "external sorting"). Can be used to sort linked lists. Can be parallelized across multiple cores. However, requires `O(n)` space for merging – other sorts don't need extra RAM. ➜ lazy person sort: get down to subs of 2 then sort then merge
Heapsort	Unstable	`O(nlogn)` always. Sometimes used in low-RAM embedded systems because of its performance/low memory requirements. ➜ heapify into max heap then remove maximum (root) and re-heapify until none left

	Binary Search Tree: ***Ordered!***	**Hash Table:** ***Unordered!***
Access	`O(log(n))`	`O(1)`
Search	`O(log(n))`	`O(1)`
Insertion	`O(log(n))`	`O(1)`
Deletion	`O(log(n))`	`O(1)`

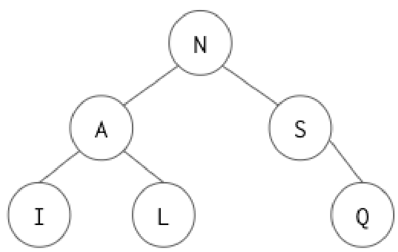
- Assuming that we are sorting numbers in increasing order:
    - One pass of Bubble Sort will move the largest item to the end.
    - One pass of Selection Sort will move the smallest item to the start.
    - After n passes of Insertion Sort, the first n items will be in sorted order (as if we completely sorted an array of size n).
    - Bubble Sort, Insertion Sort, and Selection Sort are good for simplicity.
    - Mergesort and Quicksort are good for efficiency.
    - Heapsort and Shellsort are unlikely to be on the exam: understand them and definitely bring notes.

Solution: IALNSQ

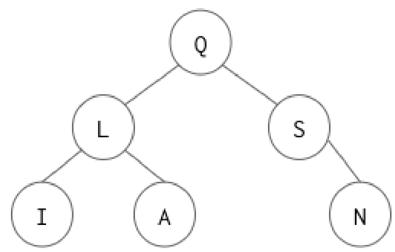
Pre-Order (CLR):



In-Order (LCR):



Post-Order (LRC):



Assuming that we are sorting numbers in increasing order:

- One pass of Bubble Sort will move the largest item to the end.
- One pass of Selection Sort will move the smallest item to the start.
- After n passes of Insertion Sort, the first n items will be in sorted order (as if we completely sorted an array of size n).
- Heapsort: Insert all N numbers into a new maxheap, While there are numbers left in the heap: Remove the biggest value from the heap, Place it in the last open slot of the array

Sorts

- Bubble Sort, Insertion Sort, and Selection Sort are good for simplicity.
- Mergesort and Quicksort are good for efficiency.
- Heapsort and Shellsort are unlikely to be on the exam: understand them and definitely bring notes.
- Priority queue: instead of the first item inserted to be first popped, pop the highest priority element
- Set, map, priority queue caveats: For user-defined classes, the set, map, and priority_queue classes all rely on some form of custom comparator to know how to order its elements:
- Unordered set, map caveats: For user-defined classes, the unordered_set and unordered_map classes rely on some form of hash function to determine its buckets; They also need to be able to determine equality between elements.
- Insertion sort is best for already sorted array O(N) or some factor of O(N).