Biostatistics 615/815 Lecture 6: Elementary Data Structures

Hyun Min Kang

September 20th, 2012

Merge Sort

Divide and conquer algorithm

- **Divide** Divide the n element sequence to be sorted into two subsequences of $n/2$ elements each
- **Conquer** Sort the two subsequences recursively using merge sort
- **Combine** Merge the two sorted subsequences to produce the sorted answer

Time complexity

- $\Theta(n \log n)$ algorithm in worst case
- Need additional memory for array copy
- In practice, slightly slower than other $\Theta(n \log n)$ algorithms due to overhead of array copy

Quicksort Algorithm

Algorithm QUICKSORT

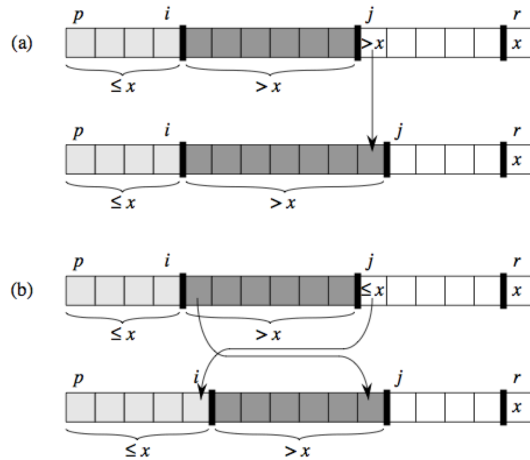
Data: array A and indices p and r
Result: $A[p..r]$ is sorted
if $p < r$ **then**
  $q = \text{PARTITION}(A,p,r)$;
  $\text{QUICKSORT}(A,p,q-1)$;
  $\text{QUICKSORT}(A,q+1,r)$;
end

Quicksort Algorithm

Algorithm PARTITION

Data: array A and indices p and r
Result: Returns q such that $A[p..q-1] \leq A[q] \leq A[q+1..r]$
$x = A[r]$;
$i = p - 1$;
for $j = p$ **to** $r - 1$ **do**
  if $A[j] \leq x$ **then**
    $i = i + 1$;
    $\text{EXCHANGE}(A[i],A[j])$;
  end
end
$\text{EXCHANGE}(A[i+1],A[r])$;
return $i + 1$;

How PARTITION Algorithm Works



Elementary data structure

Container

A container T is a generic data structure which supports the following three operation for an object x.

- SEARCH(T, x)
- INSERT(T, x)
- DELETE(T, x)

Possible types of container

- Arrays
- Linked lists
- Trees
- Hashes

Designing a simple array - myArray.h

```
#include <iostream>
#define DEFAULT_ALLOC 1024

template <class T> // template supporting a generic type
class myArray {
protected: // member variables hidden from outside
  T *data; // array of the generic type
  int size; // number of elements in the container
  int nalloc; // # of objects allocated in the memory
 public:
  myArray(); // default constructor
  ~myArray(); // destructor
  void insert(const T& x); // insert an element x, const means read-only
  bool search(const T& x);  // search for an element x and return its location
  bool remove(const T& x); // delete a particular element
  void print();  // print the content of array to the screen
};
```


Using a simple array - myArrayTest.cpp

```
#include <iostream>
#include "myArray.h"

int main(int argc, char** argv) {
  myArray<int> A;
  A.insert(10);            // {10}
  A.insert(5);             // {10,5}
  A.insert(20);            // {10,5,20}
  A.insert(7);             // {10,5,20,7}
  A.print();
  std::cout << "A.search(7) = " << A.search(7) << std::endl;   // true
  std::cout << "A.remove(10) = " << A.remove(10) << std::endl; // {5,20,7}
  A.print();
  std::cout << "A.search(10) = " << A.search(10) << std::endl; // false
  return 0;
}
```

Implementing a simple array in myArray.h

```
class myArray {
  // declarations of member variables and functions go here..
};

// If the function is not yet defined above, it can be defined as follows..
template <class T>
myArray<T>::myArray() {  // default constructor
  size = 0;              // array do not have element initially
  nalloc = DEFAULT_ALLOC;
  data = new T[nalloc];  // allocate default # of objects in memory
}

template <class T>
myArray<T>::~myArray() { // destructor
  if ( data != NULL ) {
    delete [] data;      // delete the allocated memory before destroying
  }                      // the object. otherwise, memory leak happens
}
```


myArray.h : insert

```
template <class T>
void myArray<T>::insert(const T& x) {
  if ( size >= nalloc ) {  // if container has more elements than allocated
    T* newdata = new T[nalloc*2];   // make an array at doubled size
    for(int i=0; i < nalloc; ++i) {
      newdata[i] = data[i];         // copy the contents of array
    }
    delete [] data;                 // delete the original array
    data = newdata;                 // and reassign data ptr
    nalloc *= 2;                    // double the allocation
  }
  data[size] = x;                   // push back to the last element
  ++size;                           // increase the size
}
```


myArray.h : search

```
template <class T>
bool myArray<T>::search(const T& x) {
  for(int i=0; i < size; ++i) { // iterate each element
    if ( data[i] == x ) {
      return true;
    }
  }
  return false;
}
```


myArray.h : remove

```
template <class T>
bool myArray<T>::remove(const T& x) {
  bool found = false;
  for(int i=0; i < size; ++i) { // iterate each element
    if ( data[i] == x ) { found = true; }
    if ( found && i < size-1 ) { data[i] = data[i+1]; }
  }
  if ( found ) --size;
  return found;
}
```

myArray.h : print

```
template <class T>
void myArray<T>::print() {
  if ( size > 0 ) {
    std::cout << "(" << data[0];
    for(int i=1; i < size; ++i) {
      std::cout << "," << data[i];
    }
    std::cout << ")" << std::endl;
  }
  else {
    std::cout << "(EMPTY ARRAY)" << std::endl;
  }
}
```


Implementing complex data types is not so simple

```
int main(int argc, char** argv) {
  myArray<int> A;        // creating an instance of myArray
  A.insert(10);
  A.insert(20);
  myArray<int> B = A;    // copy the instance
  B.remove(10);
  if ( ! A.search(10) ) {
    std::cout << "Cannot find 10" << std::endl; // what would happen?
  }
  return 0;              // would to program terminate without errors?
}
```


Implementing complex data types is not so simple

```
int main(int argc, char** argv) {
  myArray<int> A;        // A is empty, A.data points an address x
  A.insert(10);          // A.data[0] = 10, A.size = 1
  A.insert(20);          // A.data[0] = 10, A.data[1] = 20, A.size = 2
  myArray<int> B = A;    // shallow copy, B.size == A.size, B.data == A.data
  B.remove(10);          // A.data[0] = 20, A size = 2 -- NOT GOOD
  if ( A.search(10) < 0 ) {
    std::cout << "Cannot find 10" << std::endl; // A.data is unwillingly modified
  }
  return 0;  // ERROR : both delete [] A.data and delete [] B.data is called
}
```


How to fix it

A naive fix : preventing object-to-object copy

```
template <class T>
class myArray {
protected:
   T *data;
   int size;
   int nalloc;
   myArray(myArray& a) {}; // do not allow copying object
public:
   myArray() {...};        // allow to create an object from scratch
```

A complete fix

- `std::vector` does not suffer from these problems
- Implementing such a nicely-behaving complex object is NOT trivial
- Requires a deep understanding of C++ programming language

Summary: Array

- Simplest container
- Constant time for insertion
- $\Theta(n)$ for search
- $\Theta(n)$ for remove
- Elements are clustered in memory, so faster than list in practice.
- Limited by the allocation size. $\Theta(n)$ needed for expansion

Sorted Array

Key Idea

- Same structure with `Array`
- Ensure that elements are sorted when inserting and deleting an object
- Insertion takes longer, but search will be much faster
  - $\Theta(n)$ for insert
  - $\Theta(\log n)$ for search

Algorithms

Insert Insert the element at the end, and swap with the previous element if larger
- Same as a single iteration of INSERTIONSORT

Search Use the binary search algorithm

Remove Same as the unsorted version of `Array`

Implementation : `mySortedArray.h`

```
#define DEFAULT_ALLOC 1024
template <class T> // template supporting a generic type
class mySortedArray {
protected:     // member variables hidden from outside
   T *data;    // array of the generic type
   int size;   // number of elements in the container
   int nalloc; // # of objects allocated in the memory
   mySortedArray(mySortedArray& a) {}; // for disabling object copy
   bool search(const T& x, int begin, int end); // search with ranges
public:      // abstract interface visible to outside
   mySortedArray();         // default constructor
   ~mySortedArray();        // destructor
   void insert(const T& x); // insert an element x
   bool search(const T& x);  // search for an element x and return its location
   bool remove(const T& x); // delete a particular element
   void print();  // print the content of array to the screen
};
```


Implementation : `mySortedArrayTest.cpp`

```
#include <iostream>
#include "mySortedArray.h"

int main(int argc, char** argv) {
  mySortedArray<int> A;
  A.insert(10);           // {10}
  A.insert(5);            // {5,10}
  A.insert(20);           // {5,10,20}
  A.insert(7);            // {5,7,10,20}
  A.print();
  std::cout << "A.search(7) = " << A.search(7) << std::endl;   // true (1)
  std::cout << "A.remove(10) = " << A.remove(10) << std::endl; // true (1)
  A.print();
  std::cout << "A.search(10) = " << A.search(10) << std::endl; // false (0)
  return 0;
}
```

Constructors and destructors

```
template <class T>
mySortedArray<T>::mySortedArray() { // default constructor
  size = 0; // array do not have element initially
  nalloc = DEFAULT_ALLOC;
  data = new T[nalloc]; // allocate default # of objects in memory
}

template <class T> mySortedArray<T>::~mySortedArray() { // destructor
  if ( data != NULL ) {
    delete [] data; // delete the allocated memory before destroying
  } // the object. otherwise, memory leak happens
}
```


Implementation : `mySortedArray::insert()`

```
template <class T>
void mySortedArray<T>::insert(const T& x) {
  if ( size >= nalloc ) {  // if container has more elements than allocated
    T* newdata = new T[nalloc*2];   // make an array at doubled size
    for(int i=0; i < nalloc; ++i) {
      newdata[i] = data[i];         // copy the contents of array
    }
    delete [] data;                 // delete the original array
    data = newdata;                 // and reassign data ptr
    nalloc *= 2;                    // and double the nalloc
  }

  int i;  // scan from last to first until find smaller element
  for(i=size-1; (i >= 0) && (data[i] > x); --i) {
    data[i+1] = data[i];            // shift the elements to right
  }
  data[i+1] = x;                    // insert the element at the right position
  ++size;                           // increase the size
}
```


Implementation : `mySortedArray::search()`

```
template <class T>
bool mySortedArray<T>::search(const T& x) {
  return search(x, 0, size-1);
}

template <class T>   // simple binary search
bool mySortedArray<T>::search(const T& x, int begin, int end) {
  if ( begin > end )
    return false;
  else {
    int mid = (begin+end)/2;
    if ( data[mid] == x )
      return true;
    else if ( data[mid] < x )
      return search(x, mid+1, end);
    else
      return search(x, begin, mid-1);
  }
}
```


Implementation : `mySortedArray::remove()`

```
// same as myArray::remove()
template <class T>
bool mySortedArray<T>::remove(const T& x) {
  bool found = false;
  for(int i=0; i < size; ++i) { // iterate each element
    if ( data[i] == x ) { found = true; }
    if ( found && i < size-1 ) { data[i] = data[i+1]; }
  }
  if ( found ) --size;
  return found;
}
```

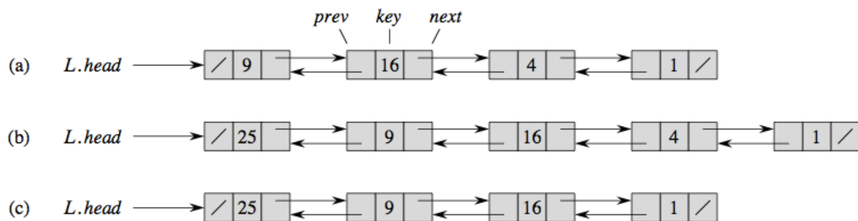
Summary: SortedArray

- $\Theta(n)$ for insertion
- $\Theta(\log n)$ for search
- $\Theta(n)$ for remove
- Optimal for frequent searches and infrequent updates
- Limited by the allocation size. $\Theta(n)$ needed for expansion

Linked List

- A data structure where the objects are arranged in linear order
- Each object contains the pointer to the next object
- Objects do not exist in consecutive memory space
  - No need to shift elements for insertions and deletions
  - No need to allocate/reallocate the memory space
  - Need to traverse elements one by one
  - Likely inefficient than `Array` in practice because data is not necessarily localized in memory
- Variants in implementation
  - (Singly-) linked list
  - Doubly-linked list

Example of a linked list



- Example of a doubly-linked list
- Singly-linked list if `prev` field does not exist

Implementation of singly-linked list

myList.h

```
#include "myListNode.h"
template <class T>
class myList {
protected:
  myListNode<T>* head; // list only contains the pointer to head
  myList(myList& a) {};    // prevent copying
public:
  myList() : head(NULL) {} // initially header is NIL
  ~myList();
  void insert(const T& x); // insert an element x
  bool search(const T& x);  // search for an element x and return its location
  bool remove(const T& x); // delete a particular element
  void print();  // print the content of array to the screen
};
```

List implementation : class myListNode

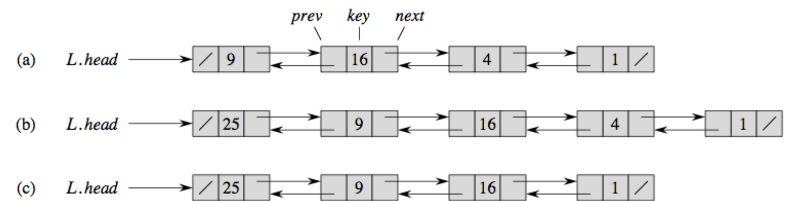
myListNode.h

```
// myListNode class is only accessible from myList class
template<class T>
class myListNode {
protected:
  T value;          // the value of each element
  myListNode<T>* next;  // pointer to the next element
  myListNode(const T& x, myListNode<T>* n) : value(x), next(n) {} // constructor
  ~myListNode();
  bool search(const T& x);
  myListNode<T>* remove(const T& x, myListNode<T>*& prevNext);
  void print(char c);
  template <class S> friend class myList; // allow full access to myList class
};
```


Inserting an element to a list

myList.h

```
template <class T>
void myList<T>::insert(T x) {
  // create a new node, and make them head
  // and assign the original head to head->next
  head = new myListNode<T>(x, head);
}
```



Destructor is required because new was used

myList.h

```
template <class T>
myList<T>::~myList() {
  if ( head != NULL ) {
    delete head;    // delete dependent objects before deleting itself
  }
}
```

myListNode.cpp

```
template <class T>
myListNode<T>::~myListNode() {
  if ( next != NULL ) {
    delete next;  // recursively calling destructor until the end of the list
  }
}
```


Searching an element from a list

myList.h

```
template <class T>
bool myList<T>::search(const T& x) {
  if ( head == NULL )  return false;  // NOT_FOUND if empty
  else return head->search(x);  // search from the head node
}
```

myListNode.cpp

```
template <class T>
// search for element x, and the current index is curPos
bool myListNode<T>::search(const T& x) {
  if ( value == x )        return true; // if found return current index
  else if ( next == NULL ) return false;     // NOT_FOUND if reached end-of-list
  else return next->search(x);  // recursive call until terminates
}
```

Removing an element from a list

myList.h

```
template <class T>
bool myList<T>::remove(const T& x) {
  if ( head == NULL )
    return false;     // NOT_FOUND if the list is empty
  else {
    // call head->remove will return the object to be removed
    myListNode<T>* p = head->remove(x, head);
    if ( p == NULL ) { // if NOT_FOUND return false
      return false;
    }
    else {             // if FOUND, delete the object before returning true
      delete p;
      return true;
    }
  }
}
```


Removing an element from a list

myListNode.h

```
template <class T>
// pass the pointer to [prevElement->next] so that we can change it
myListNode<T>* myListNode<T>::remove(T x, myListNode<T>*& prevNext) {
  if ( value == x ) {   // if FOUND
    prevNext = next;    // *pPrevNext was this, but change to next
    next = NULL;        // disconnect the current object from the list
    return this;        // and return it so that it can be destroyed
  }
  else if ( next == NULL ) {
    return NULL;        // return NULL if NOT_FOUND
  }
  else {
    return next->remove(x, next); // recursively call on the next element
  }
}
```


Summary - Linked List

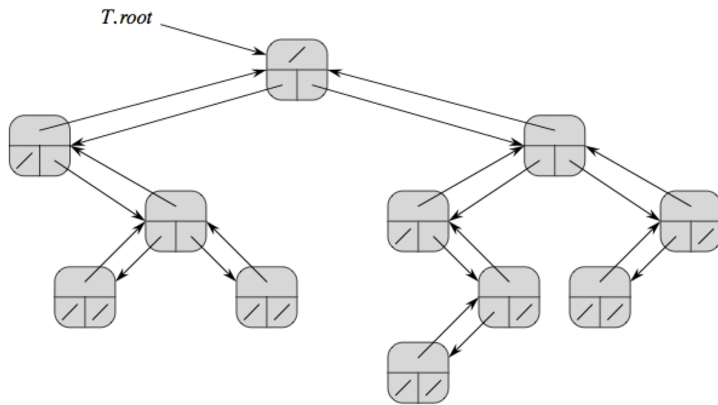
- Class Structure
  - `myList` class to keep the head node
  - `myListNode` class to store key and pointer to next node
- `Insert` algorithm : Create a new node as a head node
- `Search` algorithm
  - Return the index if key matches
  - Otherwise, advance to the next node
- `Remove` algorithm :
  - Search the element
  - Make the previous node points to the next node
  - Remove the element from the list and destroy it.
- Q: What are the advantages and disadvantages between Array and List?

Binary search tree

Data structure

- The tree contains a root node
- Each node contains
  - Pointers to `left` and `right` children
  - Possibly a pointer to its parent
  - And a key value
- Sorted : `left.key` $\leq$ `key` $\leq$ `right.key`
- Average $\Theta(\log n)$ complexity for insert, search, remove operations

An example binary search tree



Key algorithms

INSERT$(node, x)$

1. If the $node$ is empty, create a leaf node with value x and return
2. If $x < node.key$, INSERT$(node.left, x)$
3. Otherwise, INSERT$(node.right, x)$

SEARCH$(node, x)$

1. If $node$ is empty, return $-\infty$
2. If $node.key == x$, return size$(node.left)$
3. If $x < node.key$, return SEARCH$(node.left, x)$
4. If $x > node.key$, return SEARCH$(node.right, x)$ + 1 + size$(node.left)$

Key algorithms

REMOVE$(node, x)$

1. If $node.key == x$
   1. If the node is leaf, remove the node
   2. If the node only has left child, replace the current node to the left child
   3. If the node only has right child, replace the current node to the right child
   4. Otherwise, pick either maximum among left sub-tree or minimum among right subtree and substitute the node into the current node
2. If $x < node.key$
   1. Call REMOVE$(node.left, x)$ if $node.left$ exists
   2. Otherwise, return NOTFOUND
3. If $x > node.key$
   1. Call REMOVE$(node.right, x)$ if $node.right$ exists
   2. Otherwise, return NOTFOUND

Implementation of binary search tree

myTree.h

```
#include <iostream>
#include "myTreeNode.h"

template <class T>
class myTree {
protected:
  myTreeNode<T> *pRoot;    // list only contains the pointer to head
  myTree(myTree& a) {};    // prevent copying
public:
  myTree() : pRoot(NULL) {} // initially header is NIL
  ~myTree() {}
  void insert(const T& x); // insert an element x
  bool search(const T& x);  // search for an element x and return its location
  bool remove(const T& x); // delete a particular element
  void print();
};
```

Implementation of binary search tree

myTreeNode.h

```
#include <iostream>
template <class T>
class myTreeNode {
  T value;   // key value
  int size;  // total number of nodes in the subtree
  myTreeNode<T>* left;  // pointer to the left subtree
  myTreeNode<T>* right; // pointer to the right subtree

  myTreeNode(const T& x, myTreeNode<T>* l, myTreeNode<T>* r); // constructors
  ~myTreeNode();           // destructors
  void insert(const T& x); // insert an element
  bool search(const T& x);
  myTreeNode<T>* remove(const T& x, myTreeNode<T>*& pSelf);
  const T& getMax();              // maximum value in the subtree
  const T& getMin();              // minimum value in the subtree
  void print();
  template <class S> friend class myTree; // allow full access to myList class
};
```


Binary search tree : Constructors and Destructors

myTreeNode.h

```
template<class T>
myTreeNode<T>::myTreeNode(T x, myTreeNode<T>* l, myTreeNode<T>* r) :
   value(x), size(1), left(l), right(r) {}

template<class T>
myTreeNode<T>::~myTreeNode() {
  // remove child nodes before removing the node itself
  if ( left != NULL ) delete left;
  if ( right != NULL ) delete right;
}
```


Binary search tree : INSERT

myTree.h

```
template <class T>
void myTree<T>::insert(T x) {
  if ( pRoot == NULL )
     pRoot = new myTreeNode<T>(x,NULL,NULL);  // create a root if empty
  else
     pRoot->insert(x);                        // insert to the root
}
```


Binary search tree : INSERT

myTreeNode.h

```
template <class T>
void myTreeNode<T>::insert(T x) {
  if ( x < value ) {     // if key is small, insert to the left subtree
    if ( left == NULL )
      left = new myTreeNode<T>(x,NULL,NULL); // create if doesn't exist
    else
      left->insert(x);
  }
  else {                     // otherwise, insert to the right subtree
    if ( right == NULL )
      right = new myTreeNode<T>(x,NULL,NULL);
    else
      right->insert(x);
  }
  ++size;
}
```

Binary search tree : SEARCH

myTree.h

```
template <class T>
bool myTree<T>::search(const T& x) {
  if ( pRoot == NULL )
    return false;
  else
    return pRoot->search(x);
}
```


Binary search tree : SEARCH

myTreeNode.h

```
template <class T>
bool myTreeNode<T>::search(const T& x) {
  if ( x == value ) {          // if key matches to the value
    if ( left == NULL ) return true;
    else return true;
  }
  else if ( x < value ) {      // recursively call the function to left subtree
    if ( left == NULL ) return false;
    else return left->search(x);
  }
  else {
    if ( right == NULL ) return false;
    else {
      int r = right->search(x);
      if ( r < 0 ) return false;
      else if ( left == NULL ) return true;
      else return true;
    }
  }
}
```


Binary search tree : REMOVE

myTree.h

```
template <class T>
bool myTree<T>::remove(const T& x) {
  if ( pRoot == NULL ) {
    return false;
  }
  else {
    myTreeNode<T>* p = pRoot->remove(x, pRoot);
    if ( p != NULL ) {  // if an object was removed
      delete p;         // destroy the object
      return true;      // and return true
    }
    else {
      return false;     // return false if the object was not found
    }
  }
}
```


Binary search tree : REMOVE

myTreeNode.h

```
template <class T>
myTreeNode<T>* myTreeNode<T>::remove(const T& x, myTreeNode<T>*& pSelf) {
  if ( x == value ) {  // key was found
    if ( ( left == NULL ) && ( right == NULL ) ) { // no child
      pSelf = NULL; return this;
    }
    else if ( left == NULL ) { // only left is NULL
      pSelf = right; right = NULL; return this;
    }
    else if ( right == NULL ) {  // only right is NULL
      pSelf = left; left = NULL; return this;
    } // ....
```

Binary search tree : REMOVE (cont'd)

myTreeNode.h

```
    else { // neither left nor right is NULL
      // choose which subtree to delete
      myTreeNode<T>* p;
      if ( left->size > right->size ) { // if left subtree is larger
        const T& m = left->getMax();       // copy the largest value among them
        p = left->remove(m, left);  // to current node, and delete the node
        value = m;
      }
      else {
        const T& m = right->getMin();       // copy smallest value among them
        p = right->remove(m, right); // to current node, and delete the node
        value = m;
      }
      return p;
    }
  }
```


Binary search tree : REMOVE (cont'd)

myTreeNode.h

```
  else if ( x < value ) {
    if ( left == NULL ) return NULL;
    else return left->remove(x, left);
  }
  else { // x > value
    if ( right == NULL ) return NULL;
    else return right->remove(x, right);
  }
}
```


Binary search tree : GETMAX and GETMIN

myTreeNode.h

```
template <class T>
const T& myTreeNode<T>::getMax() {  // return the largest value
  if ( right == NULL ) return value;
  else return right->getMax();
}

template <class T>
const T& myTreeNode<T>::getMin() {  // return the smallest value
  if ( left == NULL ) return value;
  else return left->getMin();
}
```


If you want to print a tree...

myTreeNode.h

```
template <class T> void myTreeNode<T>::print() {
  std::cout << "[ ";
  if ( left != NULL ) left->print();
  else std::cout << "[ NULL ]";
  std::cout << " , (" << value << "," << size << ") , ";
  if ( right != NULL ) right->print();
  else std::cout << "[ NULL ]";
  std::cout << " ]";
}
```

myTree.h

```
template <class T> void myTree<T>::print() {
  if ( pRoot != NULL ) pRoot->print();
  else std::cout << "(EMPTY TREE)";
  std::cout << std::endl;
}
```

Summary - Binary Search Tree

- Key Features
  - Fast insertion, search, and removal
  - Implementation is much more complicated
- Class Structure
  - `myTree` class to keep the root node
  - `myTreeNode` class to store key and up to two children
- Key Algorithms
  - Insert : Traverse the tree in sorted order and create a new node in the first leaf node.
  - Search : Divide-and-conquer algorithms
  - Remove : Move the nearest leaf element among the subtree and destroy it.

Two types of containers

Containers for single-valued objects - last lecture

- $\text{INSERT}(T, x)$ - Insert x to the container.
- $\text{SEARCH}(T, x)$ - Returns the location/index/existence of x.
- $\text{REMOVE}(T, x)$ - Delete x from the container if exists
- STL examples include `std::vector`, `std::list`, `std::deque`, `std::set`, and `std::multiset`.

Containers for `(key,value)` pairs - this lecture

- $\text{INSERT}(T, x)$ - Insert $(x.key, x.value)$ to the container.
- $\text{SEARCH}(T, k)$ - Returns the value associated with key k.
- $\text{REMOVE}(T, x)$ - Delete element x from the container if exists
- Examples include `std::map`, `std::multimap`, and `__gnu_cxx::hash_map`

Direct address tables

An example `(key,value)` container

- $U = \{0, 1, \cdots, N-1\}$ is possible values of keys (N is not huge)
- No two elements have the same key

Direct address table : a constant-time container

Let $T[0, \cdots, N-1]$ be an array space that can contain N objects

- $\text{INSERT}(T, x) : T[x.key] = x$
- $\text{SEARCH}(T, k) : \text{RETURN } T[k]$
- $\text{REMOVE}(T, x) : T[x.key] = \text{NIL}$

Analysis of direct address tables

Time complexity

- Requires a single memory access for each operation
- $O(1)$ - constant time complexity

Memory requirement

- Requires to pre-allocate memory space for any possible input value
- $2^{32} = 4GB\times$(size of data) for 4 bytes (32 bit) key
- $2^{64} = 18EB(1.8 \times 10^7 TB)\times$(size of data) for 8 bytes (64 bit) key
- An infinite amount of memory space needed for storing a set of arbitrary-length strings (or exponential to the length of the string)

Hash Tables

Key features

- $O(1)$ complexity for INSERT, SEARCH, and REMOVE
- Requires large memory space than the actual content for maintaining good performance
- But uses much smaller memory than direct-address tables

Key components

- Hash function
  - $h(x.key)$ mapping key onto smaller 'addressible' space H
  - Total required memory is the possible number of hash values
  - Good hash function minimize the possibility of key collisions
- Collision-resolution strategy, when $h(k_1) = h(k_2)$.

Summary

Today

- List
- Binary Search Tree
- Direct Address Table
- Introduction to hash table

Next Lecture

- More hash tables
- Dynamic programming