Biostatistics 615/815 Lecture 7: Data Structures

Hyun Min Kang

Januray 27th, 2011

Announcements

Another good and bad news

Announcements

Another good and bad news

- Homework #2 is finally announced
- Due is on Feb 7th, but better start earlier

Announcements

Another good and bad news

- Homework #2 is finally announced
- Due is on Feb 7th, but better start earlier

815 projects

- Instructor will send out E-mails to individually later today.

Recap : Radix sort

Key idea

- Sort the input sequence from the last digit to the first repeatedly using a linear sorting algorithm such as CountingSort
- Applicable to integers within a finite range

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Recap : Elementary data structures

	SEARCH	INSERT	DELETE
Array	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
SortedArray	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
List	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Hash	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

- Array or list is simple and fast enough for small-sized data
- Tree is easier to scale up to moderate to large-sized data
- Hash is the most robust for very large datasets

Recap : Memory management with user-defined data type can be tricky

```
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) < 0 ) {
    std::cout << "Cannot find 10" << std::endl; // what would happen?
  }
  return 0;              // would to program terminate without errors?
}
```

Today

More data structures

- Sorted array
- Linked list
- Binary search tree
- Hash table

Focus

- Key concepts
- How to implement the concepts to working examples
- But not too much detail of advanced C++

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

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

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

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

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

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

```
// Exactly the same as myArray.h
#include <iostream>
#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 genetic type
   int size;   // number of elements in the container
   int nalloc; // # of objects allocated in the memory
   mySortedArray(mySortedArray& a) {}; // for disabling object copy
   int 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
   int search(const T& x);  // search for an element x and return its location
   bool remove(const T& x); // delete a particular element
};
```

Implementation : Main.cpp

```
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}
  std::cout << "A.search(7) = " << A.search(7) << std::endl;   // returns 1
  std::cout << "A.search(10) = " << A.search(10) << std::endl; // returns 2
  mySortedArray<int>& B = A; // copy is disallowed but reference is allowed
  std::cout << "B.search(10) = " << B.search(10) << std::endl; // returns 2
  return 0;
}
```

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>
int mySortedArray<T>::search(const T& x) {
  return search(x, 0, size-1);
}
template <class T>   // simple binary search
int mySortedArray<T>::search(const T& x, int begin, int end) {
  if ( begin > end )
    return -1;
  else {
    int mid = (begin+end)/2;
    if ( data[mid] == x )
      return mid;
    else if ( data[mid] < x )
      return search(x, mid+1, end);
    else
      return search(x, begin, mid);
  }
}
```

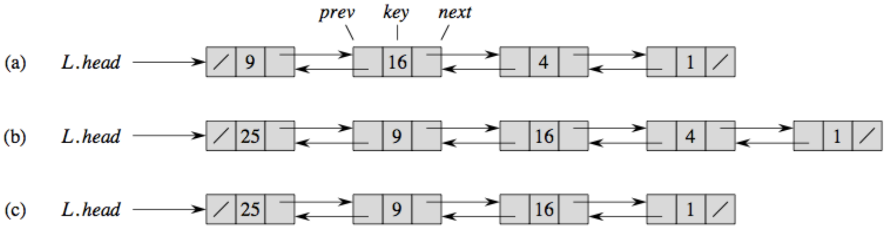
Implementation : mySortedArray::remove()

```
// same as myArray::remove()
template <class T>
bool mySortedArray<T>::remove(const T& x) {
  int i = search(x);  // try to find the element
  if ( i >= 0 ) {       // if found
    for(int j=i; j < size-1; ++j) {
      data[i] = data[i+1];  // shift all the elements by one
    }
    --size;            // and reduce the array size
    return true;       // successfully removed the value
  }
  else {
    return false;      // cannot find the value to remove
  }
}
```

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
  int search(const T& x);  // search for an element x and return its location
  bool remove(const T& x); // delete a particular element
};
```

List implementation : class myListNode

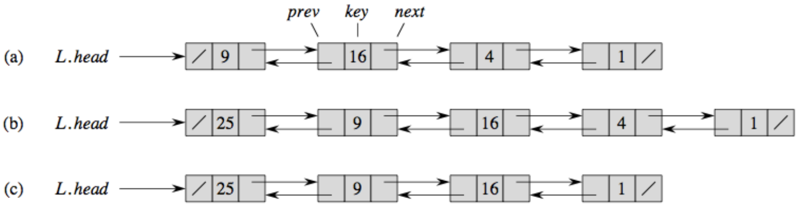
myListNode.h

```
template<class T>
class myListNode {
  T value;          // the value of each element
  myListNode<T>* next;  // pointer to the next element
  myListNode(const T& v, myListNode<T>* n) : value(v), next(n) {} // constructor
  ~myListNode();
  int search(const T& x, int curPos);
  myListNode<T>* remove(const T& x, myListNode<T>** pPrevNext);
  template <class S> friend class myList; // allow full access to myList class
};
```

Inserting an element to a list

myList.cpp

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



Destroying a list object

myList.cpp

```
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.cpp

```
template <class T>
int myList<T>::search(const T& x) {
  if ( head == NULL )  return -1;  // NOT_FOUND if empty
  else return head->search(x, 0);  // search from the head node
}
```

myListNode.cpp

```
template <class T>
// search for element x, and the current index is curPos
int myListNode<T>::search(const T& x, int curPos) {
  if ( value == x )          return curPos; // if found return current index
  else if ( next == NULL ) return -1;      // NOT_FOUND if reached end-of-list
  else return next->search(x, curPos+1);  // recursive call until terminates
}
```

Removing an element from a list

myList.cpp

```
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.cpp

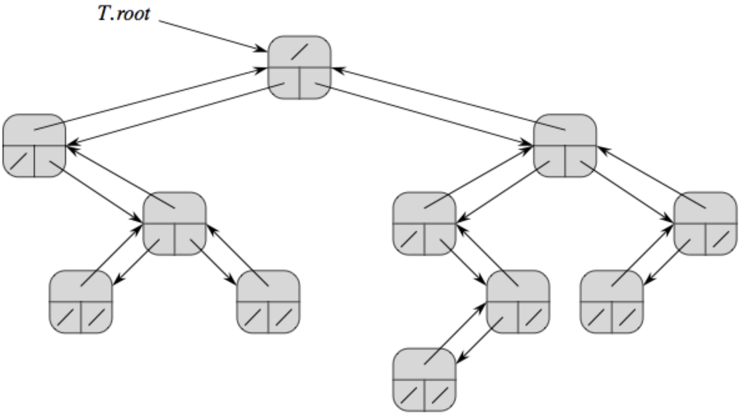
```
template <class T>
// pass the pointer to [prevElement->next] so that we can change it
myListNode<T>* myListNode<T>::remove(const T& x, myListNode<T>** pPrevNext) {
  if ( value == x ) {   // if FOUND
    *pPrevNext = next;  // *pPrevNext was this, but change to next
    next = NULL;        // disconnect 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
  }
}
```

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

$\textsc{Remove}(node, x)$

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 $\textsc{Remove}(node.left, x)$ if $node.left$ exists
   2. Otherwise, return NOTFOUND
3. If $x > node.key$
   1. Call $\textsc{Remove}(node.right, x)$ if $node.right$ exists
   2. Otherwise, return NOTFOUND

Implementation of binary search tree

myTree.h

```
template <class T>
class myTree {
protected:
  myTreeNode<T>* pRoot;       // tree contains pointer to root
  myTree(myTree& a) {};       // prevent copying
public:
  myTree() { pRoot = NULL; } // initially root is empty
  void insert(const T& x);
  int search(const T& x);
  bool remove(const T& x);
};
```

Implementation of binary search tree

myTreeNode.h

```
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
  int search(const T& x);
  myTreeNode<T>* remove(const T& x, myTreeNode<T>** ppSelf);
  T getMax();                // maximum value in the subtree
  T getMin();                // minimum value in the subtree
};
```

Binary search tree : INSERT

myTree.cpp

```
template <class T>
void myTree<T>::insert(const 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.cpp

```
template <class T>
void myTreeNode<T>::insert(const 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.cpp

```
template <class T>
int myTree<T>::search(const T& x) {
  if ( pRoot == NULL )
    return -1;
  else
    return pRoot->search(x);
}
```

Binary search tree : SEARCH

myTreeNode.cpp

```
template <class T>   // return the 0-based rank of the value x
int myTree<T>::search(const T& x) {
  if ( x == value ) {          // if key matches to the value
    if ( left == NULL )
      return 0;                // return 0 if there is no smaller element
    else
      return left->size;       // return # of left-subtree otherwise
  }
  else if ( x < value ) {      // recursively call the function to left subtree
    if ( left == NULL )
      return -1;
    else
    return left->search(x);
  }
```

Binary search tree : SEARCH

myTreeNode.cpp (cont'd)

```
  else { // when x > value, [#leftSubtree]+1 should be added
    if ( right == NULL )
      return -1;
    else {
      int r = right->search(x);
      if ( r < 0 ) return -1;
      else if ( left == NULL ) return ( 1 + r );
      else return ( left->size + 1 + r );
    }
  }
}
```

Today

Elementary data structures

- Sorted array
- Linked list
- Binary search tree

Next Lecture

- Hash tables
- Dynamic Programming