The exam will be drawn entirely from the following questions. There will be no "surprise" or "trick" questions on the exam. Every question on the exam is in this handout except that some questions may be phrased differently or involve different data.

Exam 1 Questions

1. Every question from Exam 1. From the exam itself, **not** from the exam review questions.

Dynamic Sets

2. Fill in the following table of "Big-Oh" worst-case time performance for the given dynamic-set operations using the given implementations:

Operation	Array List	Singly Linked List	Sorted Array List	Sorted Singly Linked List	BST
Search					
Insert					
Append					
Delete					
Empty					
Output in Ascending Order					
Output in Descending Order					

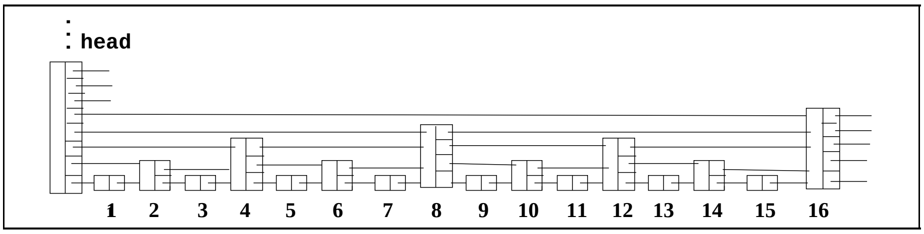
Skip Lists

Please see the definitions of `SkipListNode` and `SkipList` on pages 8 and 9.

3. What is a skip list?

4. The following perfect skip-list is valid for $p = \frac{1}{2}$. Draw an equivalent figure for $p = \frac{1}{4}$. What distribution of node levels do you expect in a long list of this type?

5. The expected asymptotic time performance for `SkipList` operations is $O(\lg n)$. There is a non-zero probability that the performance could be as bad as $O(n)$. Draw a 7 element `SkipList`, with `int` data values, that would have such poor performance. Use a maximum node level of 4.



6. What maximum node size is appropriate for a `SkipList` suitable for storing up to 65,536 elements and with associated probability $\frac{1}{4}$.

7. Write pseudo-code for the `Search` operation in a skip-list. Given a search item, return `true` if the item is in the list, `false` otherwise.

8. Given a skip list (a drawing of one), indicate all the comparisons done in searching for a particular element.

9. Suppose you have a skip list that uses probability p and suppose it contains N nodes. Show the expected distribution of node sizes (*i.e.*, the number of nodes at each size).

Hash Tables

10. Name three desirable properties of a good hash function.

11. What is a "collision" in a hash table?

12. What is the "clustering" problem in hash tables?

13. Describe the "division" and "multiplication" methods of generating hash values.

14. Describe the "separate chaining" and "open addressing" collision resolution strategies.

15. What do the terms "linear probing" and "quadratic probing" mean?

16. Describe "double hashing."

17. Hash tables do not support efficient sequential access of the elements in sorted order. Nonetheless, write pseudo-code that performs that operation in a linear-open-addressed table and in a separate-chaining table.

18. An open-addressing hash function is a function:

$$h : U \times \{0, 1, \ldots\} \rightarrow \{0, 1, \ldots, m-1\}$$

where m is the size of the hash table. Using a hash-table size of 10, and $U = \{89, 18, 49, 58, 69\}$, write a hash function that uses linear probing. Use that hash function to generate the indices for the elements of U. Do the same using quadratic-probing.

19. A double-hashing function can be written in the form

$$h(k,i) = (h_1(k) + ih_2(k)) mod\, m$$

where k is a key, i is a hash attempt, m is the size of the hash-table, and h_1 and h_2 are ordinary hash functions. If $h_1(k) = k mod\, m$, write a suitable $h_2(k)$.

20. Suppose a double-hashing scheme uses the hash functions

$$h_1(k) = k mod\, 10$$

$$h_2(k) = 7 - (k mod\, 7)$$

$$h(k,i) = (h_1(k) + ih_2(k)) mod\, 10$$

When the hash table size is 10, what indices do the keys {89,18,49,58,69} hash to?

Priority Queues and Heaps

Please see the definition of the `Heap` class on page 11.

21. Define "binary heap."

22. Define "leftist heap."

23. Let n be the number of nodes in a binary tree. For what values of n is a complete binary tree also a full binary tree? For what values of n is a complete binary tree also a perfect binary tree?

24. Insertion and deletion in a binary heap is in $O(\lg n)$. Explain why this is so.

25. Finding the minimum element in a binary heap (a min-heap) is in $O(1)$. Explain why this is so.

26. Prove that the largest element in a min heap is a leaf.

27. Describe (pseudo-code is fine) an algorithm to find the largest element in a binary min-heap. What is the asymptotic worst case performance of your algorithm?

28. A binary heap contains the following elements in the order shown:

$$2, 8, 3, 10, 16, 7, 18, 13, 15$$

Show the order that results at each stage of inserting the element 4.

29. A binary heap contains the following elements in the order shown:

$$2, 8, 3, 10, 16, 7, 18, 13, 15$$

Show the order that results at each stage of deleting the minimum element.

30. Describe the process for constructing a binary heap from a given set of initial values (*i.e.*, heapify an array of elements). A pseudo-code description is fine.

31. Construct a binary heap using the initial values

$$18, 2, 13, 10, 15, 3, 7, 16, 8$$

Show the heap at each stage of construction.

32. Prove that a binary heap can be constructed from n initial values in $O(n)$ time.

33. Given a drawing of a binary tree, state if it is a leftist tree and if it is a leftist heap. If not, state why not.

34. Prove that any complete binary tree is a leftist tree.

35. Prove: for any leftist tree having N vertices, the number of vertices, R, on the rightmost path to a non-full vertex is given by

$$R \leq \lfloor \lg(N + 1) \rfloor$$

36. Sort an array by using heapsort.

37. Define `FindMin` for a leftist heap.

38. Using the `Meld` operation, define `Insert` and `DeleteMin` operations for leftist heaps.

39. Prove that two leftist heaps of sizes n_1 and n_2, respectively, can be melded (merged) in $O(\lg n_1 + lgn_2)$ time.

40. Give pseudocode for constructing a leftist heap from a given set of values. Your code must construct a leftist heap of N elements in $O(N)$ time (**not** in $O(n \lg n)$).

AVL Trees

41. Define *AVL tree*.

42. Show the result of inserting `2,1,4,5,9,3,6,7` into an empty AVL tree (show the tree at the end of each insertion).

43. Show the result of deleting a given node in the tree.

44. What is the "Big-Oh" performance (in terms of the number of nodes in the tree) for each operation `Search`, `Insert`, and `Delete` for AVL trees in the best, worst, and average cases?

45. What property of AVL trees is most significant in explaining their "Big-Oh" behavior for the operations `Search`, `Insert`, and `Delete`.

Red-Black Trees

46. Define *Red-Black tree*.

47. Show the result of inserting `2,1,4,5,9,3,6,7` into an empty Red-Black tree (show the tree at the end of each insertion).

48. Show the result of deleting a given node in the tree.

49. What is the "Big-Oh" performance (in terms of the number of nodes in the tree) for each operation `Search`, `Insert`, and `Delete` for Red-Black trees in the best, worst, and average cases?

50. What property of Red-Black trees is most significant in explaining their "Big-Oh" behavior for the operations `Search`, `Insert`, and `Delete`.

51. Prove that a red-black tree with n internal vertices has height $h \leq 2\lg(n+1)$

Splay Trees

52. Define *Splay tree*.

53. Show the result of inserting `2,1,4,5,9,3,6,7` into an empty Splay tree (show the tree at the end of each insertion).

54. Show the result of deleting a given node in the tree.

55. What is the "Big-Oh" performance (in terms of the number of nodes in the tree) for each operation `Search`, `Insert`, and `Delete` for Splay trees in the best, worst, and average cases?

56. What property of Splay trees is most significant in explaining their "Big-Oh" behavior for the operations `Search`, `Insert`, and `Delete`.

57. What does the following statement mean?

    > "a splay tree has $O(n \lg n)$ amortized performance over sequences of `Insert`, `Delete`, and `Search` operations"

58. Given that the `Search` operation

```
template <class T>
Bnode<T>* BST<T>::Search(Bnode<T>* u, int key, Boolean& success) const;
```

returns the node contining `key` if successful and the node last encountered if unsuccessful:

   (a) Modify the BST `Insert` method given on page 13 to work for a splay tree.

   (b) Modify the BST `Predecessor` method given on page 14 to work for a splay tree.

   (c) Write a `Delete` method for splay tree.

Definition of SkipListNode class

```
template <class T>
class SkipListNode
{
private:
  int _size;  // number of levels in this node
  Array<SkipListNode *> _forward;  // forward pointers
  T datum;  // datum stored at this node
protected:
 inline void SetDatum(T & datum);
 inline void SetForward(int i, SkipListNode<T> * f);
 inline void SetSize(int sz);
public:
 SkipListNode();
 SkipListNode(int size);
 SkipListNode(T & data, int size);
 SkipListNode(const SkipListNode<T> &);
 ~SkipListNode();

 inline T & GetDatum();
 inline int GetSize();
 inline SkipListNode<T> * GetForward(int level);

 friend class SkipList<T>;
};
```

Definition of SkipList class

```
template <class T>
class SkipList
{
private:
  int _high_node_size;  // highest-level node now in this list
  int _max_node_size;  // highest-level node allowed in this list
  double prob;  // probability associated with this list
  SkipListNode<T> * _head;  // header node of this list
protected:
  // Search for item in this skiplist, starting at startnode
  // startnode will usually be the header.  If item is found,
  // return the node at which it is found with success equal true.
  // Otherwise, return the preceding node with success equal false.
  // Precondition:  startnode is not NULL
  SkipListNode<T> * Search(T & item, SkipListNode<T> * startnode);

  // Accessor for _head
  inline SkipListNode<T> * GetHeader();

  // Finds the point at which item would be inserted using a node
  // of the given size.
  // Returns a "lookback" node that has each of its forward
  // pointers set to the closest node at that level.
  SkipListNode<T> * FindInsertPoint(T & item, int nodesize)

  // insert item into this skiplist, using a node of the
  // specified size. Return true if successful
  bool Insert(T & item, int nodesize);
```

SkipList Definition continued on next page

Continuation of SkipList Definition

```
public:
  SkipList(); // use max_node_size of 16, prob of 0.25
  SkipList(int max_node_size, double probab);
  SkipList(const SkipList &);
  ~SkipList();

  int GetHighNodeSize(); // size of node with most levels

  // insert item into this skiplist.  Return true if insertion
  // succeeds, false otherwise.
  bool Insert(T & item);

  // Search for item in skiplist.  If found, return the item with
  // success equal true.  Otherwise, return a dummy item with
  // success equal false.
  T & Search(T & item, bool & success);

  // delete the first occurrence of the specified item
  bool Delete(T & item);
};
```

Definition of `Heap` class

```
template <class T>
class Heap
{
protected:
  virtual void SetLength(int len) = 0;

  // Sifts up the element at index i in this heap
  virtual void SiftUp(int index) = 0;

  // Sifts down the element at index i in this heap
  virtual void SiftDown(int index) = 0;
public:
  // Adds element x to this heap.
  // Return: true if insertion succeeds, false otherwise.
  virtual bool Insert(const T& x) = 0;

  // Returns the minimum element in this heap.
  // It is an error to attempt this operation on an empty heap.
  virtual T& FindMin() = 0;

  // Deletes the minimum element in this heap.
  // Return: true if deletion succeeds, false otherwise.  Deletion
  // from an empty heap returns false.
  virtual bool DeleteMin() = 0;

  // Accessor for the size of this heap.
  // Return: the number of elements on this heap.
  virtual int GetLength() const = 0;

  // Test for this heap being empty.
  // Return: true if this heap is empty, false otherwise.
  bool IsEmpty() const;
```

Heap Definition continued on next page

Continuation of Heap Definition

```
  // Makes this heap be empty
  virtual void MakeEmpty() = 0;

  // A string describing this heap.  For example, the string
  // describing a Binary Heap priority queue would be "Binary Heap"
  // Return: a string description
  virtual char * ToString() = 0;

  // Accessor for an iterator over this heap.  Outputs the
  // heap elements in level order
  // Return: ptr to the Iterator.
  // Note: dynamically allocated.  User is responsible for
  // storage deallocation.
  virtual Iterator<T> * LevelOrderIterator() = 0;
};
```

The BST::Insert Operation from Heileman

```
template<class T>
Boolean BST<T>::Insert(T item)
{
  Boolean found;
  Bnode<T>* new_node = new Bnode<T>(item);
  assert(new_node != 0);
  if (root == 0)
    root = new_node;
  else {
   Bnode<T>* v = Search(root, item.Key(), found);
   if (found == true) {  // item is already in tree
       delete new_node;
       return false;
   }
   if (item.Key() < v->data.Key())
     BinaryTree<T>::InsertLeaf(new_node, v, left);
   else
     BinaryTree<T>::InsertLeaf(new_node, v, right);
 }
 return true;
}
```

The `BST::Predecessor` Method from Heileman

```
template<class T>
T& BST<T>::Predecessor(T & item, bool& success) const
{
  Bnode<T>* v = Search(root, item, success);

  if(success == false)
    return *(new T);  // unsuccessful search, return dummy value

  if (v->lchild != 0)
    return SubtreeMax(v->lchild)->data;

  Bnode<T>* original = v;
  Bnode<T>* p = v->parent;
  while (p != 0 && v == p->lchild)
    {
      v = p;
      p = p->parent;
    }

  if (p == 0)
    return *(new T); // no predecessor, return dummy value

  return p->data;
}
```