CPSC 535: Advanced Algorithms

Instructor: Dr. Doina Bein

Fundamental Data Structures

- Short review of them
- The open content book available at http://opendatastructures.org/ covers them in detail
- PDF version: http://opendatastructures.org/ods-python.pdf
- HTML version: http://opendatastructures.org/ods-python/
- The book is geared towards a Python implementation of these data structures, but the fundamentals are the same, independent of the implementation
- Other websites about teaching introductory computer science algorithms, including searching, sorting, recursion, and graph theory:

https://www.khanacademy.org/computing/computer-science/algorithms

Abstract Data Types (ADT)

- An abstract data type (ADT) is a set of objects that are related to each other together with a set of operations:
  - Description of the objects that compose the data type
  - Description of the relationships between the individual objects
  - Description of the operations to be performed on the objects
- An ADT is a *mathematical abstraction*: nowhere in its definition is any mention of *how* the operations are implemented
- Familiar examples of data types: integers, reals, booleans, arrays.
- Examples of ADT: lists, sets, graphs, trees, along with their operations

ADT versus Its Implementation

- The familiar examples of data types are built-in in programming language (you do not need to worry about their implementation; it is already done)
- An ADT is language-independent and must be implemented in an appropriate computer language
- Difference between an abstract data type (ADT) and the implementation: how each ADT functions and not how is implemented
- An ADT may be implemented in several ways using the same programming language
- If an implementation is done correctly, the program that use them does not necessarily need to know which implementation was used => *information hiding*

Operations on Data Types

- Data type Integers: addition (+), subtraction (-), multiplication (*), integer division (/), comparisons (<, <=, >, >=, ==, !=) etc.
- Data type Reals: addition (+), subtraction (-), multiplication (*), division (/), comparisons (<, <=, >, >=, ==, !=) etc.
- Data Type Booleans: and (&&), or (||), not (!).
- Data Type Array: fetch and store using []

Array ADT

- An ***array*** is a sequence indexed by a system of integer coordinates
- The system of indexes is used to access elements and to permit alteration of individual elements.
- The elements can be accessed directly in Θ(1) time (constant time)
- The type array is built in any high level programming language, but we can still define our own type array and implement it in a different way
- Two integers m and n, $m:n$ denotes all integers in the range $m..n$
- In a k–dimensional array A, the index i_j in the dimension j ranges between $m_j:n_j$: $A[i_1, i_2, \ldots i_k]$ is the element of array A with index $(i_1, i_2, \ldots i_k)$
- To store a k-dimensional array $A[m_1:n_1,\ m_2:n_2, \ldots,\ m_k:n_k]$ in memory the obvious way is to store it using consecutive memory cells

- A storage allocation function $\mathtt{Loc}(\mathtt{A}[\mathtt{i}_1,\mathtt{i}_2,\ldots\mathtt{i}_k])$ is a mapping function that assigns a distinct memory address to each distinct array index $(\mathtt{i}_1,\mathtt{i}_2,\ldots\mathtt{i}_k)$ of array $\mathtt{A}$
- Each element of the array can occupy L memory cells; for example, an integer needs four (4) memory cells
- For a linear (1-dimensional) array X(0:6), the first element X[0] is stored at location X, the second element X[1] is stored at location X+L, the i-th element is stored at location $X + L \cdot i$
- For multi-dimensional arrays that whose elements have the same size (the same type), the most popular storage allocation methods: lexicographic (or row-major order) and column-major order

Lexicographic (Row-Major) Storage Allocation

- A lexicographic ordering of the integer k-tuples $(i_1, i_2, \cdots, i_k)$ in the set $m_1 : n_1 \times m_2 : n_2 \times \cdots m_k : n_k$ is defined as $(a_1, a_2, \cdots, a_k) < (b_1, b_2, \cdots, b_k)$ iff there is some j in the range $1 \leq j \leq k$ such that $a_i = b_i$ for all $1 \leq i <$ j and $a_j < b_j$
- Example: the indices of a 2-dimensional array A[0:2,0:3] in lexicographic order are
  $(0,0) < (0,1) < (0,2) < (0,3) < (1,0) < (1,1) < (1,2) < (1,3) < (2,0) < (2,1) < (2,2) < (2,3)$
- Example: the indices of a 3-dimensional array A[0:2,0:3,0:2] in lexicographic order are
  (0,0,0) < (0,0,1) < (0,0,2) < (0,1,0) < (0,1,1) < (0,1,2) < (0,2,0) < (0,2,1) < (0,2,2) < (0,3,0) < (0,3,1) < (0,3,2) < (1,0,0) < (1,0,1) < (1,0,2) < (1,1,0) < (1,1,1) < (1,1,2) < (1,2,0) < (1,2,1) < (1,2,2) < (1,3,0) < (1,3,1) < (1,3,2) < (2,0,0) < (2,0,1) < (2,0,2) < (2,1,0) < (2,1,1) < (2,1,2) < (2,2,0) < (2,2,1) < (2,2,2) < (2,3,0) < (2,3,1) < (2,3,2)

- If we display the elements of A[0:2,0:3] as a rectangular array (see below) then lexicographic storage (see right) means that first row 0 of A is stored in locations 0:3, then row 1 of A is stored in locations 4:7, and finally row 2 of A is stored in locations 8:11. Thus the name *row-major order*.

A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[1,0]	A[1,1]	A[1,2]	A[1,3]
A[2,0]	A[2,1]	A[2,2]	A[2,3]

Location	Array element
0	A[0,0]
1	A[0,1]
2	A[0,2]
3	A[0,3]
4	A[1,0]
5	A[1,1]
6	A[1,2]
7	A[1,3]
8	A[2,0]
9	A[2,1]
10	A[2,2]
11	A[2,3]

- Given an arbitrary array `A[2:7, 3:9]`, `Loc(A[4,5])` can be calculated as follows:

$$Loc(A[4,5]) = Loc(A[2,3]) + row\ 2 + row\ 3 + L \cdot (5 - 3 + 1)$$

where means $row\ 2$ the number of memory cells to store the entire row 2 which is $L \cdot (9 - 3 + 1)$. Thus

$$Loc(A[4,5]) = Loc(A[2,3]) + (4 - 2) \cdot L \cdot (9 - 3 + 1) + L \cdot (5 - 3 + 1)$$

- In general, given a 2-dimensional array $A[m_1 : n_1, m_2 : n_2\ ]$, the formula for row-major order storage allocation of

$$Loc(A[i_1, i_2\ ]) = Loc(A[m_1, m_2])$$
$$+L \cdot \big((i_1 - m_1) \cdot (n_2 - m_2 + 1) + (i_2 - m_2)\big)$$

- Or $Loc(A[i_1, i_2\ ]) = Loc(A[m_1, m_2\ ]) + c_0 + c_1 \cdot i_1 + c_2 \cdot i_2$ where

$$c_2 = L$$
$$c_1 = L \cdot (n_2 - m_2 + 1)$$
$$c_0 = -L \cdot m_1 \cdot (n_2 - m_2 + 1) - L \cdot (m_2 - 1)$$

- General formula for row-major order storage allocation of an array $\mathrm{A}[m_1 : n_1, m_2 : n_2, \cdots, m_k : n_k]$

$$Loc(A[i_1, i_2, \cdots, i_k]) = Loc(A[m_1, m_2, \cdots, m_k]) + c_0 + c_1 \cdot i_1 + c_2 \cdot i_2 + \cdots + c_k \cdot i_k$$

for suitable constants c_i $(1 \leq i \leq k)$

- Row-major order allocation is efficient:
  - Traversal of elements in a direction parallel to any arbitrary axis of coordinate system is easy and can usually be implemented using machine index registers
  - Access to an arbitrary element can be computed using the formula given above

Column-major Storage Allocation

- If we display the elements of A[0:2,0:3] as a rectangular array (see below) then *column-major order* storage (see right) means that first column 0 of A is stored in locations 0:2, then column 1 of A is stored in locations 3:5, then column 2 of A is stored in locations 6:8, and finally column 3 of A is stored in locations 9:11.

A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[1,0]	A[1,1]	A[1,2]	A[1,3]
A[2,0]	A[2,1]	A[2,2]	A[2,3]

Location	Array element
0	A[0,0]
1	A[1,0]
2	A[2,0]
3	A[0,1]
4	A[1,1]
5	A[2,1]
6	A[0,2]
7	A[1,2]
8	A[2,2]
9	A[0,3]
10	A[1,3]
11	A[2,3]

- Column-major order is different from row-major (lexicographic) order since, for instance, A[2,0] appears earlier in column-major order than A[0,1] even though A[0,1] comes before A[2,0] in row-major (lexicographic) order
- In general, given a 2-dimensional array $\mathrm{A}[m_1 : n_1, m_2 : n_2\ ]$, the formula for column-major order storage allocation of

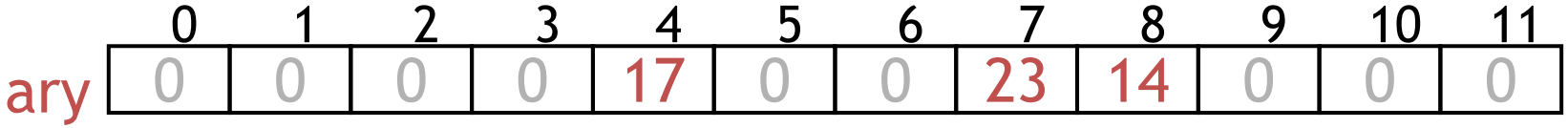
$$Loc(A[i_1, i_2\ ]) = Loc(A[m_1, m_2])$$
$$+L \cdot \big((i_2 - m_2) \cdot (n_1 - m_1 + 1) + (i_1 - m_1)\big)$$

- Two-dimensional arrays are stored in column-major order in FORTRAN

Sparse Array ADT

- A *sparse array* is simply an array most of whose entries are nil (or 0 or some other default value)
- Example: Suppose you wanted a 2-dimensional array of course grades, whose rows are CSUF students and whose columns are courses
- There are about 45,000 students (a fact)
- There are about 3,000 courses (assumption, I could not find the number of courses)
- This array would have about 135,000,000 entries
- Since most students take fewer than 3000 courses, there will be a lot of empty spaces in this array
- This is a big array, even by modern standards
- There are ways to represent sparse arrays efficiently

- We will start with sparse one-dimensional arrays, which are simpler



- We could represent it as a linked list

- We need to get values from the array by operation *fetch*

```
<element type> fetch(int index)
```

which searches a linked list for a given index

- And to store values in the array by operation *store*

```
void store(int index, <element type> value)
```

which searches a linked list for a given index and inserts the element in the list

Implement the Sparse Array ADT in Java

```
class List {
   int index;      // the index number
   Object value;     // the actual data
   List next;// the "pointer"
   public Object fetch(int index) {
      List current = this;  // first "List" (node) in the list
      do {
         if (index == current.index) {
              return current.value;  // found correct location
         }
         current = current.next;
     } while (index < current.index && next != null);
     return null;          // if we get here, it wasn't in the list
   }
}
```

- The store operation is basically the same, with the extra complication that we may need to insert a node into the list

- Although “stepping through an array” is not a fundamental operation on an array, it is one we do frequently

  ```
  for (int i = 0; i < array.length; i++) {...}
  ```

- We have a list, and all we need to do is step through it.
- Expand the ADT by adding operations
- An interface, in Java, is like a class, but
  - It contains only public methods (and maybe some final values)
  - It only declares methods; it doesn’t define them
- Example:

```
public interface Iterator { // Notice: no method bodies
    public boolean hasNext( );
    public Object next( );
    public void remove( );
}
```

- This is an interface that is defined for you, in java.util

- Example:

```
public class SparseArrayIterator implements Iterator {
   private List current; // pointer to current cell in the list
   SparseArrayIterator(List first) { // the constructor
       current = first;
   }
  public boolean hasNext() {
      return current != null;
   }
  public Object next()  {
      Object value = current.value;
      current = current.next
      return value;
  };
  public void remove() {
       // We don't want to implement this, so...
       throw new UnsupportedOperationException();
  }
}
```

Data Structures

Data Structure: method for storing, organizing data

- Data members e.g. head pointer, tail pointer
- Invariant(s) defining how the structure must be organized to remain valid, e.g. head points to first node, tail points to last node
- Defined operations, each operation is an algorithm that operates on the structure.

Dynamic Sets

- Sets that can change over time are called *dynamic*
- A *dictionary* is a dynamic set that supports the following operations: insert elements into, delete elements from, test membership in a set.
- Other operations can be supported
- Each element in a set is represented by an *object* whose fields can be examined and manipulated
- Some types of dynamic sets assume that one of the object's field is an identifying *key* field (aka key)
- Some types of dynamic sets assumes that the keys are drawn from a totally ordered set (e.g. real numbers, all words in a dictionary under the usually alphabetic ordering)

Operations on Dynamic Sets

- The operations can be grouped into two types:
  - Queries: return info about the set, an element in the set, or a group of elements in the set
  - Modifying operations: change the set
- Typical operations for a set S:
  - `Search(S, k)`: given `S` and a key value `k`, return a pointer `x` to an element in `S` whose key is `k` (`key[x]=k`) or `NIL`, if `S` does not contain such an element (query)
  - `Insert(S,x)`: augment `S` with the element pointed by `x`, assuming that all the fields in `x` have been initialized (modifying op.)
  - `Delete(S,x)`: given a pointer `x` to an element of `S`, remove `x` from `S` (modifying op.). Note: `x` is a pointer and not a key value. (modifying op.)
  - `Minimum(S,x)`: given a totally ordered set `S`, return the element of `S` with the smallest key value (query)

Operations on Dynamic Sets

- `Maximum(S,x)`: given a totally ordered set `S`, return the element of `S` with the largest key value (query)
- `Successor(S,x)`: given an element `x` whose key is from a totally ordered set `S`, return the next larger element in `S`, or `NIL` if `x` is the maximum element (query)
- `Predecessor(S,x)`: given an element `x` whose key is from a totally ordered set `S`, return the next smaller element in `S`, or `NIL` if `x` is the minimum element (query)

- Successor and Predecessor can be extended to sets with non-distinct keys

Priority Queue

- A priority queue is a list (an ADT) that maintains `S` elements, each with an associated value called a *key*
- A max-priority queue supports the following operations:
  - `Init(S)` : Initialize the priority queue to be empty.
  - `Is_empty(S)` : Test to see whether the priority queue is empty.
  - `Insert(S,x)` : inserts the elements x into the set S. It can be written $S=S\cup\{x\}$
  - `Maximum(S)` : returns the element of S with the largest key
  - `Extract-max(S)`: removes and returns the element of S with the largest key
  - `Increase-key(S,x,k)` : increases the value of element `x`'s key to the new value `k`, which is assumed to be at least as large as `x`'s current key value

Priority Queue

- Stacks, queue, heaps (min heap and max heap) are examples of a priority queue:
  - In a stack, the priority item is the most recently inserted item (called the *top*).
  - In a queue, the priority item is the least recently inserted item (insert at *front*; peek and delete from *rear*)
  - In a heap, the priority item is the item with the minimum (or maximum) key. The key could be anything.

Example: The items of a PQ are unfulfilled obligations.

- The Bill Payer problem: You get bills in the mail from time to time, and you have to pay them. But you don't pay bills the moment you receive them: instead, you place them on your desk, where they are unfulfilled obligations. When you decide to pay a bill, you take one from your desk and pay it.
- *The stack strategy*: When you get a bill, you place it on the top of your pile of bills. When you decide to pay a bill, you pay the one on top, i.e., the one you got most recently.
- *The queue strategy*: The bill you pay is the one that's been on your desk the longest. One way to do this is to insert every new bill at the bottom (rear) of the pile, and take from the top (front) of the pile.
- *The heap strategy*: Every bill has a due date. When you decide to pay, you pay the bill that has the earliest due date. How would you implement this in practice?

Stack (aka LIFO queue) operation	Worst Case
Create an empty stack	O(1)
Push an element into the stack	O(1)
Return the top element of the stack	O(1)
Pop and return the top of the stack	O(1)
Test whether the stack is empty	O(1)

Can I do the following operations?

- Return the bottom of the stack. Answer: NO
- Test whether an element is in the stack. Answer:
    - YES: I can only test the top element
    - NO: I cannot check the entire stack

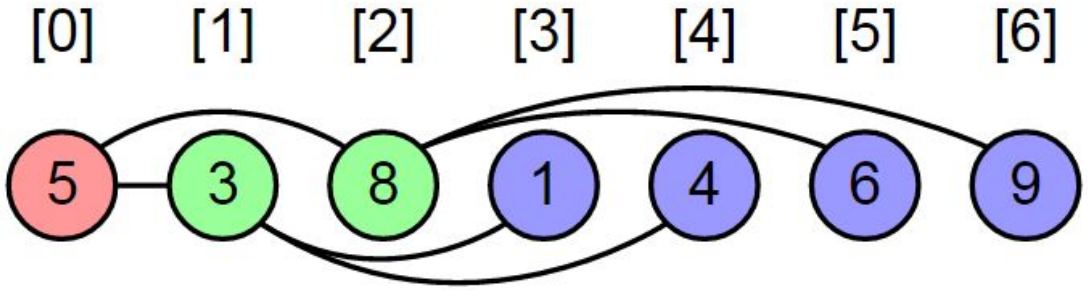
Queue (FIFO queue) operation	Worst Case
Create an empty queue	O(1)
Push an element into the queue	O(1)
Return the rear of the queue	O(1)
Pop the queue	O(1)
Pop and return the rear of the queue	O(1)
Test whether the queue is empty	O(1)

Can I do the following operations?

- Return the front of the queue. Answer: NO. (Technically, I can do it since I have access to the front of the queue, but it violates the property of a queue ADT, when only the element at the rear of the queue can be returned.)
- Test whether an element is in the queue. Answer:
    - YES: I can only test the rear element
    - NO: I cannot check the entire queue

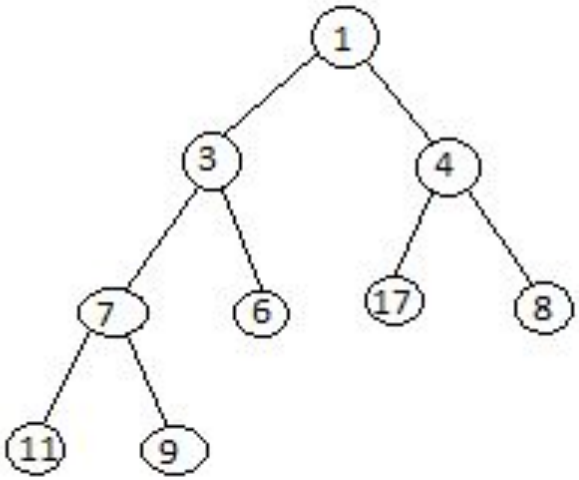
Heaps

- A (binary) heap is a *complete* binary tree and the nodes in the tree follow a *heap-order property*.
- A *complete* binary tree is completely filled except maybe the lowest level, which is filled from left to right.
- A complete binary tree can be represented as an array : array `A` has `length[A]` elements but only the indices `0.. heapsize[A]` are used for the heap.



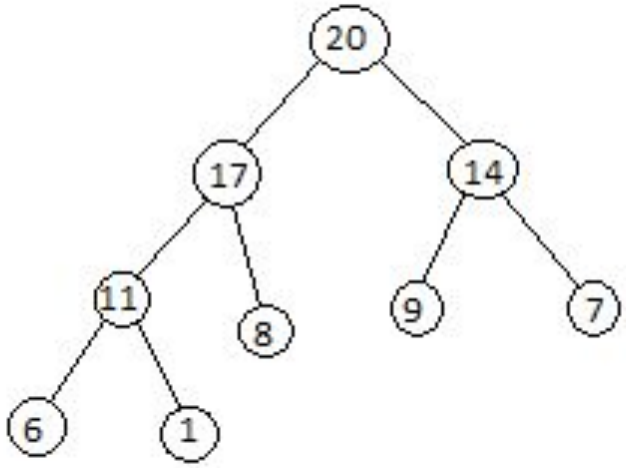
- The root of the tree is stored at `A[0]`
- Given the index `i` of a node:
  - `A[(i-1)/2]` stores the parent of `i`, `Parent(i)`
  - `A[2*i+1]` stores `i`'s left child `Left(i)` and
  - `A[2*i+2]` stores `i`'s right child `Right(i)`

- *Heap-order property*:
  - A tree follows the *heap-order* property if it follows either the max-heap or the min-heap property.
  - A tree follows the *max-heap* order property if the value stored at every node `x` is ≤ the value stored at the parent of `x`, except the root which has no parent: `A[Parent(i)] ≥ A[i]` or `A[(i-1)/2] ≥ A[i]`
  - A tree follows the *min-heap* property if the value stored at every node `x` is ≥ the value stored at the parent of `x`, except the root which has no parent: `A[Parent(i)] ≤ A[i]` or `A[(i-1)/2] ≤ A[i]`



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

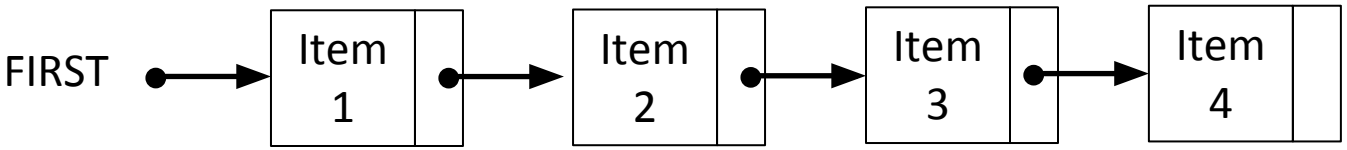
Figures taken from
http://www.studytonight.com/data-structures/heap-sort

- Maintaining the Heap Property for max-heaps:
  - If an element `A[i]` violates the max-heap property w.r.t. its left child i.e. `A[i] < A[2*i+1]` and/or w.r.t its right child `A[i] < A[2*i+2]` but the subtrees rooted at `Left(i)` and `Right(i)` are max-heaps, then `A[i]` needs to be push down the tree

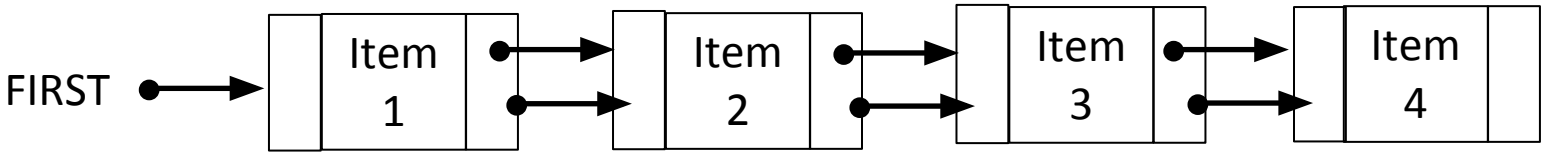
Min-heap operation	Worst Case
Create an empty heap	O(1)
Convert an unsorted list into a heap in-place	O(n)
Push an element into the heap	O(log n)
Find the minimum element into the heap	O(1)
Pop the minimum element out of the heap & restructure heap	O(log n)
Return the top of the heap (aka the minimum element)	O(1)
Pop, return the top of the heap & restructure heap	O(log n)
Test whether the heap is empty	O(1)

Linked Lists

- A list is a finite sequence of items drawn from some set
- The simple linked lists are singly or doubly:
- A *singly linked list* is a list in which each element points to its successor; a list item stores an element and a pointer to its successor



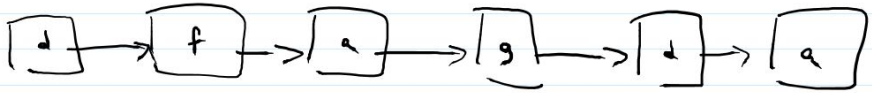
- In a *doubly linked list*, each element points to its successor and to its predecessor; a list item stores an element and two pointers, one to its successor and one to its predecessor



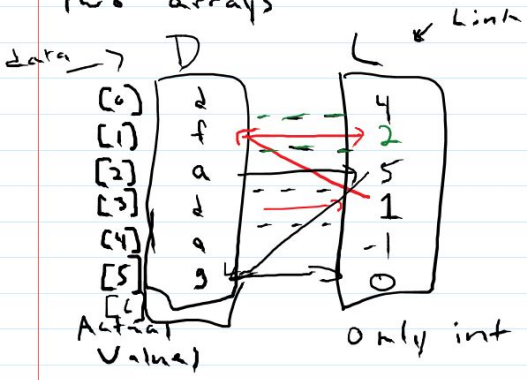
Implementing the list using arrays

- Given a list of length n, two arrays D and L, of length n can be used to implement the list and the list operations
- The array D will contain the elements in the list
- The array L will contain the index of the subsequent elements
- For a position i, $\mathrm{D}[i]$ will store some element in the list and $\mathrm{L}[i]$ will store the index in the array D of the element following $\mathrm{D}[i]$ in the list
- Examples

List: d, f, a, g, d, a



Two arrays



head = 3 (the first element is stored in index 3)

D[head] = first element in the list

L[head] = index of the next element

(D[i], L[i]) = a node in the linked list

head = 3

L[3] = 1 → next element is stored at index 1

L[1] = 2 L[5] = 0 L[4] = -1

L[2] = 5 L[0] = 4

List: m, p, g, a, d, b

	D	L
[0]	a	3
[1]	b	-1
[2]	g	0
[3]	d	1
[4]	p	2
[5]	m	4

head = 5

L[head] = index of D of next element

L[L[head]] = index of next next

L[L[L[head]]] = index in D of the 4th element

Any type of data that can be stored in the list can be stored in D

Only ints can be stored in L

List operation	Worst Case
Create an empty list	O(1)
Test whether the list is empty	O(1)
Insert an element in the list at the head	O(1)
Insert an element anywhere besides the head	O(n)
Insert an element at the end when the tail is maintained	
Insert an element at the end when the tail is not maintained	O(n)
Search for an element based on its value (key)	O(n)
Delete an element in the list based on its value (key)	O(n)

The delete operation is based on which operation?
(Answer: search)

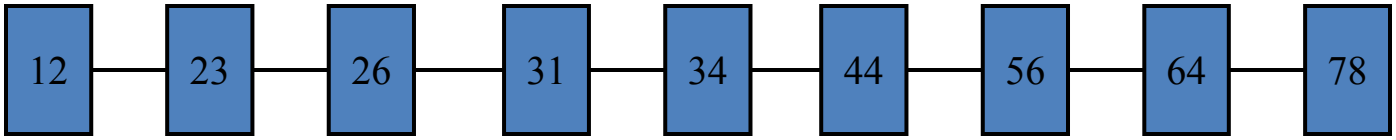
Skips Lists

(taken from http://u.cs.biu.ac.il/~amir/DS/w3_skip_lists_biu.ppt and https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec12.pdf)

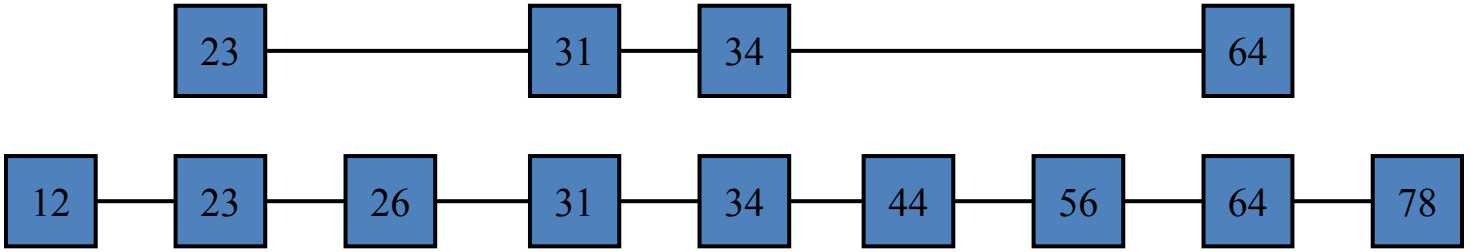
- A *skip list* is a data structure for dictionaries that uses a randomized insertion algorithm
  - Invented by William Pugh in 1989
- In a skip list with n entries
  - The expected space used is O(n)
  - The expected search, insertion and deletion time is O(log n)
- Skip lists are fast and simple to implement in practice

From list to skip list

- Start from simplest data structure: (sorted) linked list
- Search takes Θ(n) time in worst case
- How can we speed up searches?

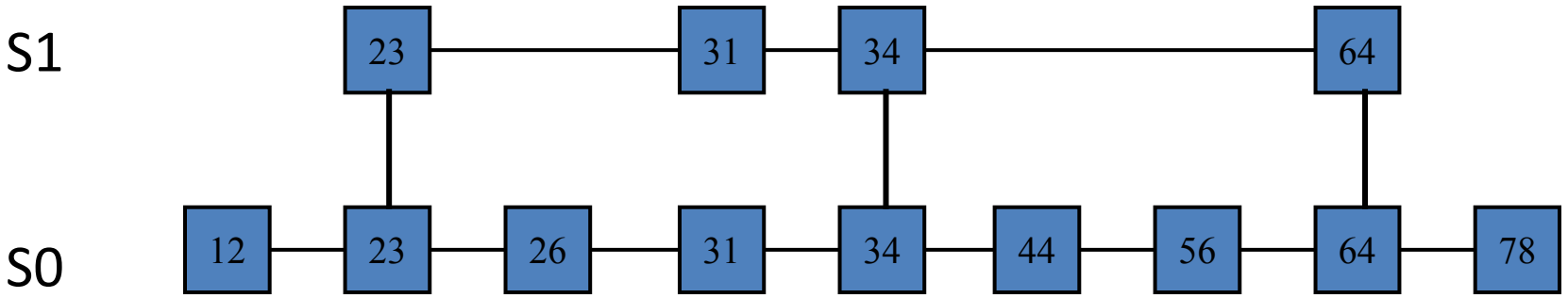


- Suppose we had two sorted linked lists (on subsets of the elements)
- Each element can appear in one or both lists • How can we speed up searches?



Linked lists as subway lines

- IDEA: Express and local subway lines (à la New York City 7th Avenue Line)
  - Express line connects a few of the stations
  - Local line connects all stations
  - Links between lines at common stations



Searching in two linked lists

SEARCH(x):

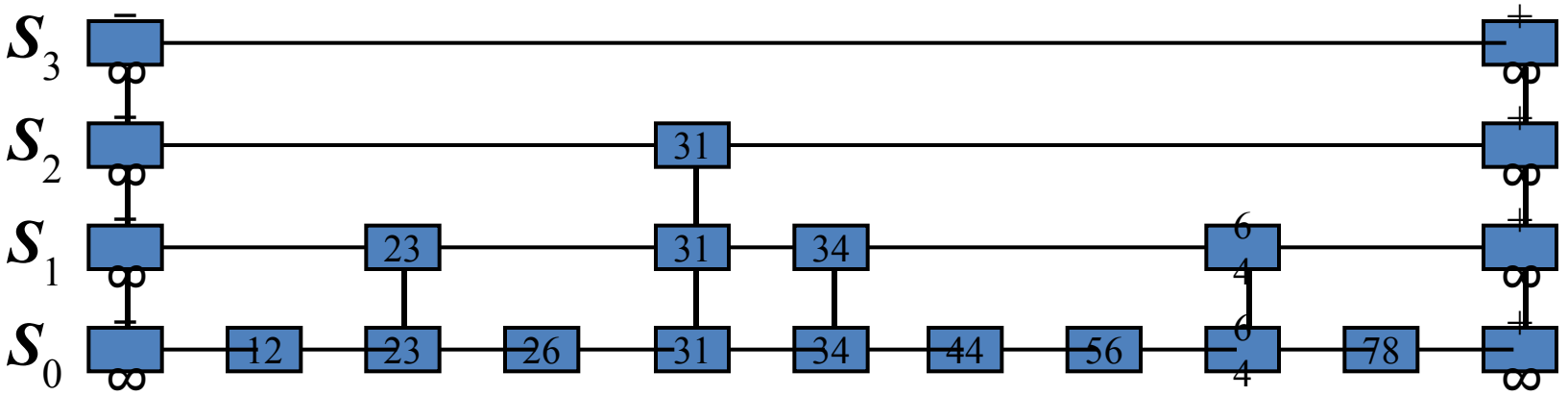
- Start with topmost list (S1) and walk right until until going right would go too far
- Walk down to lower linked list (S0) and walk right in S0 until element found (or not)

Choosing what nodes go to S1

- QUESTION: Which nodes should be in S1?
  - In a subway, the "popular stations"
  - Here we care about worst-case performance
    - Best approach: Evenly space the nodes in S1
    - But how many nodes should be in S1?
- ANALYSIS:
  - Search cost is roughly |S1| + |S1|/|S0|
  - Minimized (up to constant factors) when terms are equal: $|S1|^2 = |S0| = n \Rightarrow |S1| = \sqrt{n}$
  - Search cost is thus roughly $2\sqrt{n}$

More linked lists?

- What if we had more sorted linked lists?
- 2 sorted lists $\Rightarrow 2\sqrt{n}$
- 3 sorted lists $\Rightarrow 3\sqrt[3]{n}$
- k sorted lists $\Rightarrow \mathrm{k}\sqrt[k]{n}$
- lg n sorted lists $\Rightarrow \lg \mathrm{n}\ \sqrt[\lg n]{n} = 2\lg \mathrm{n}$
- lg n sorted linked lists are like a binary tree (in fact, level-linked B+-tree)

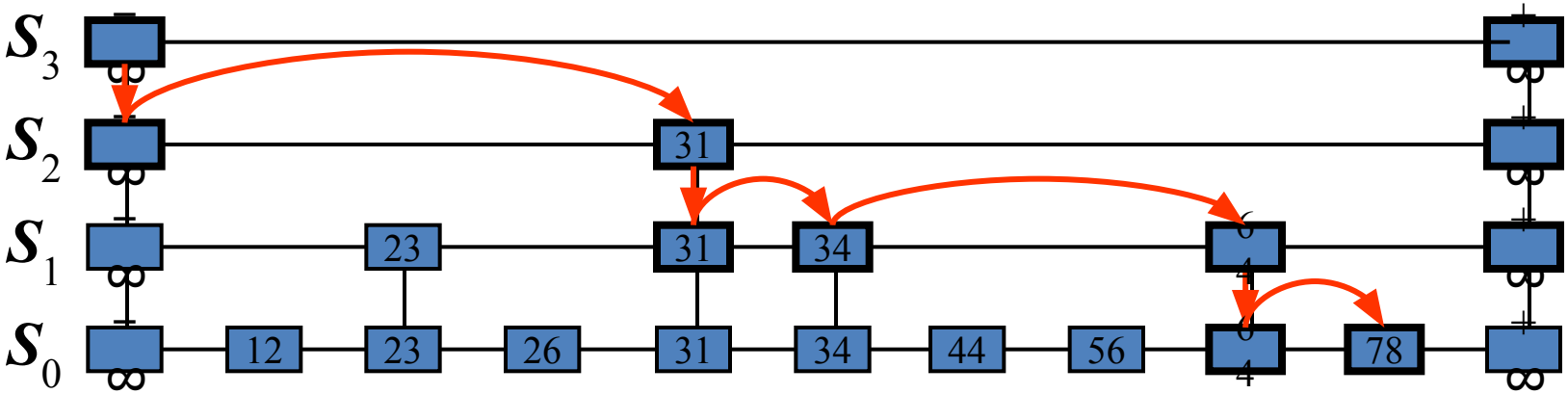


Skip Lists

- A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in nondecreasing order
  - Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$
  - List S_h contains only the two special keys
- We show how to use a skip list to implement the dictionary ADT

Search

- We search for a key x in a a skip list as follows:
  - We start at the first position of the top list
  - At the current position p, we compare x with $y \leftarrow key(next(p))$
    - $x == y$: we return $element(next(p))$
    - $x > y$: we "scan forward"
    - $x < y$: we "drop down"
  - If we try to drop down past the bottom list, we return *null*
- Example: search for 78



Randomized Algorithms

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution
- It contains statements of the type

```
b ← random()
if b = 0
  do A ...
else { b = 1}
  do B ...
```

- Its running time depends on the outcomes of the coin tosses
- We analyze the expected running time of a randomized algorithm under the following assumptions
  - the coins are unbiased, and
  - the coin tosses are independent
- The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give "heads")
- We use a randomized algorithm to insert items into a skip list

Insert(x)

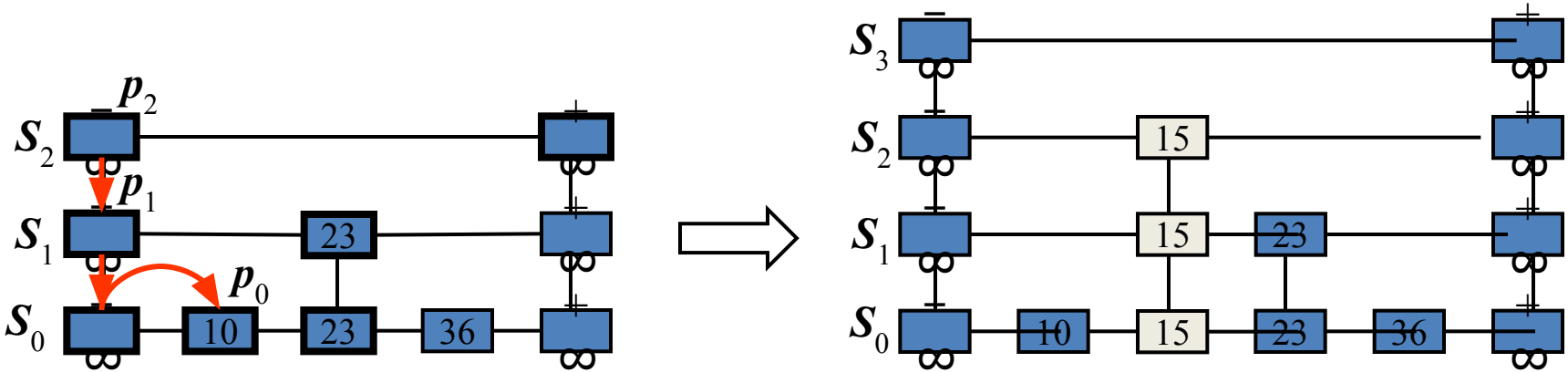
QUESTION: To which other lists should we add x?

IDEA:

- Flip a (fair) coin; if HEADS, promote x to next level up and flip again to see if x will be inserted in upper lists
- Let i be the number of consecutive HEADS
- When inserting x in a skip list, we search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
- For $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j
- Probability of promotion to next level = 1/2
- On average:
  - 1/2 of the elements promoted 0 levels
  - 1/4 of the elements promoted 1 level
  - 1/8 of the elements promoted 2 levels
  - etc.

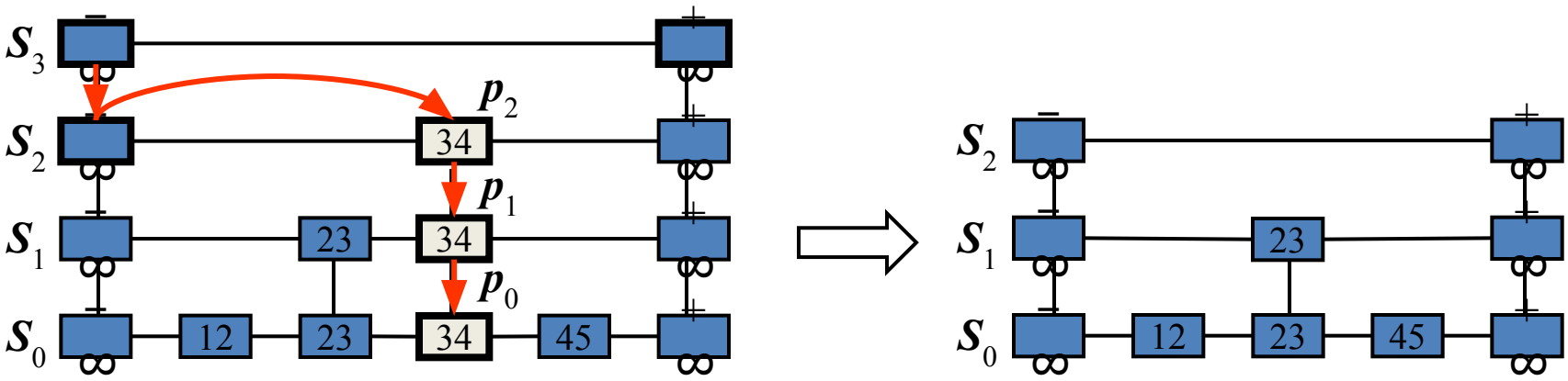
Example

- Example: insert key 15, with HEADS, HEADS, HEADS, i.e. $i = 2$



Delete(x)

- To remove an entry with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions p_0, p_1, …, p_i of the items with key x, where position p_j is in list S_j
  - We remove positions p_0, p_1, …, p_i from the lists S_0, S_1, … , S_i
  - We remove all but one list containing only the two special keys
- Example: remove key 34



Summary

- A skip list is the result of insertions (and deletions) from an initially empty structure (containing just $+\infty$ and $-\infty$)
  - INSERT(x) uses random coin flips to decide promotion level
  - DELETE(x) removes x from all lists containing it
- THEOREM: With high probability, every search in an n-element skip list costs O(lg n)

Space Usage

- The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm
- We use the following two basic probabilistic facts:

  Fact 1: The probability of getting i consecutive heads when flipping a coin is $1/2^i$

  Fact 2: If each of n entries is present in a set with probability p, the expected size of the set is np

- Consider a skip list with n entries
  - By Fact 1, we insert an entry in list S_i with probability $1/2^i$
  - By Fact 2, the expected size of list S_i is $n/2^i$
- The expected number of nodes used by the skip list is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

- Thus, the expected space usage of a skip list with n items is $O(n)$

Read more

- Read more at https://courses.csail.mit.edu/6.046/spring04/handouts/skiplists.pdf

Array ADT & Matrices

- An ***array*** is a sequence indexed by a system of integer coordinates
- The system of indexes is used to access elements and to permit alteration of individual elements.
- The elements can be accessed directly in O(1) time (constant time)
- The type array is built in any high level programming language, but we can still define our own type array and implement it in a different way
- Two integers `m` and `n`, `m:n` denotes all integers in the range `m..n`
- A matrix is a two-dimensional array but it can be extended to an arbitrary dimension
- In a `k`–dimensional array `A`, the index i_j in the dimension `j` ranges between $m_j : n_j$: $A[i_1, i_2, \ldots i_k]$ is the element of array `A` with index $(i_1, i_2, \ldots i_k)$

Matrix operation	Worst Case
Initialize a matrix with $r \times c$ elements with value x	O(r*c)
Return number of rows	O(1)
Return number of columns	O(1)
Lookup element at row i and column j	O(1)
Set a value for element at row i and column j	O(1)