Chapter 2
Some Elementary Data Structures and Algorithms

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

- Linus Torvalds

References:

[CLRS 2.1, 10.1-10.3]

[SSS 3.1-3.2, 4.1-4.2, 4.9]

Data Structures

- We need efficient (time and space) ways to represent the data we are to process
- Common operations on data
  - Insert, delete
  - Search
  - Arrange in order
- Common data structures:
  - Array
  - Linked list
  - Stack
  - Queue

Data Structures

- Why different data structures?
  - Support different operations with different running times
  - Different algorithms require different number of each type of operations
- Algorithms/data structures are inter-dependent
  - “Algorithms + Data Structures = Programs” [N. Wirth 1976]

Array

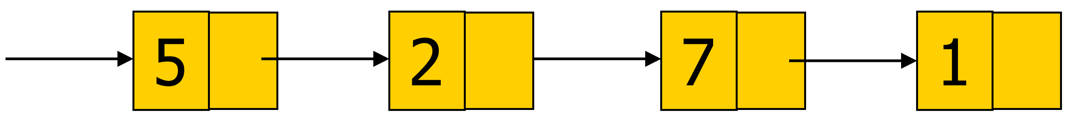
- Simplest, most common data structure
- Contiguous memory locations
- *Direct access* – A[i] in O(1) time
- Search: read the entire array, worst case O(n) time
- Insert:
  - At the end: easy (but may need memory allocation)
  - At specific position: need to shift elements after that position, O(n) time
- Delete:
  - Shift elements, O(n) time

5	2	7	1

5	2	4	7	1

Linked List

- Can we support insert/delete efficiently?
- *Linked list:* memory locations linked by pointers (references)



Memory

0		
1	5	6
2		
3	1	/
4	7	3
5		
6	2	4

  - Directly available in Java
  - But we need to understand how it works to analyse its running time

Linked List

- In C/C++ styled pseudocode:

```
struct node {
  int data;    // data
  node *next; // pointer
}

node *head = new node; // head of new list
```

Linked List: Search

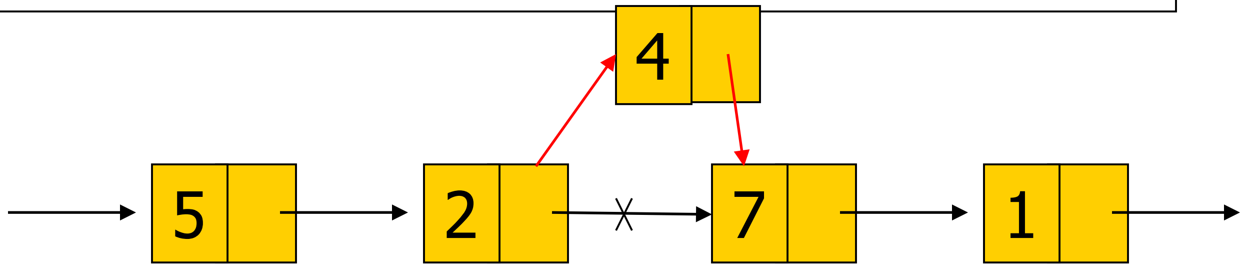
- Linked list does not support direct access!
  - To get the i-th element, you need to go through the list $\rightarrow$ i steps
- Search for an element:
  - O(n) time

```
node* search(int x) {
  node *temp = head;
  while (temp != null && temp->data != x)
    temp = temp->next;
  return temp; // null if not found
}
```

Linked List: Insert

- Insert at a specific position:
  - If we already have a pointer to that position, then just modify pointers
  - O(1) time

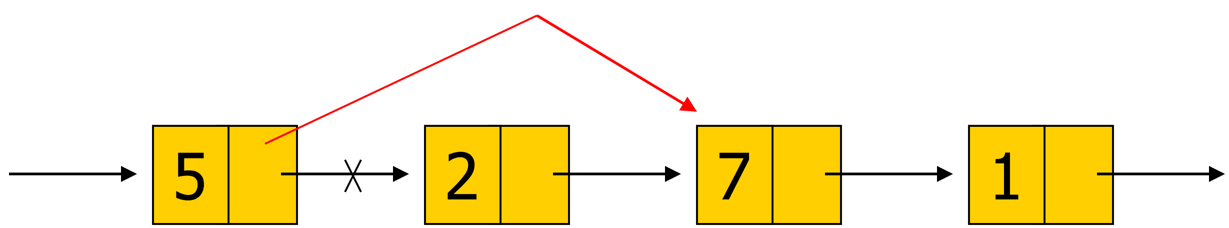
```
// insert number x after node y
void insert(int x, node* y) {
  node t = new node;
  t->num = x;
  t->next = y->next;
  y->next = t;
}
```



Linked List: Delete

- Delete an element
  - Suppose we have a reference to the element (if not, search for it)
  - O(1) time

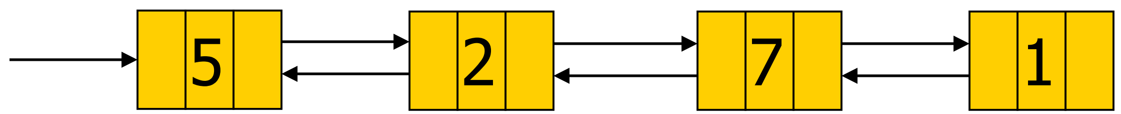
```
// delete element pointed to by x (after x)
void delete(node x) {
  x->next = x->next->next;
}
```



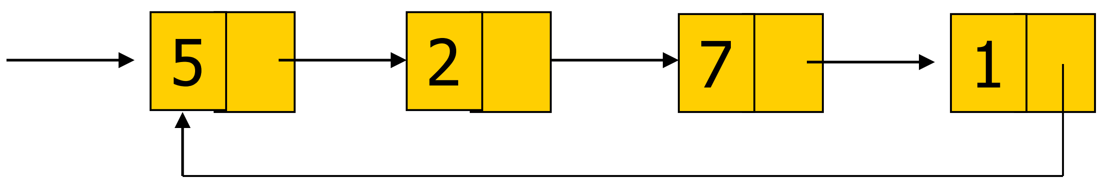
  - (we ignored boundary conditions in these insert/delete code)

Linked List: Variations

- *Doubly linked list:*
  - Pointers to previous and next element



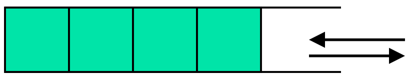
- *Circular linked list:*
  - Last element point to first one



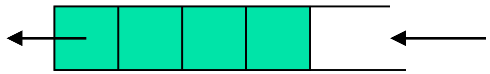
- Support easier navigation through the list

Stacks and Queues

- *Stack:* last-in-first-out (LIFO)



  - Insert and delete at the same end (top)
  - Operations:
    - Push(v): insert v to stack S
    - Pop(): return and remove top element from S
    - Top(): return (not remove) top element in S
- *Queue:* first-in-first-out (FIFO)



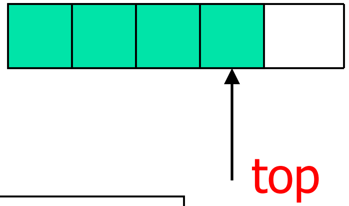
  - Insert at one end (tail), delete at other end (head)
  - Operations:
    - Enqueue(v): insert v into queue Q
    - Dequeue(): return and remove first element from Q

Abstract Data Structures

- Stacks and queues are *abstract data structures*
  - Implemented using other data structures
- Both stack or queue can be implemented using array or linked list
- Support constant time per operation
- In the following we discuss implementing them using arrays
  - Assume a maximum limit on size known

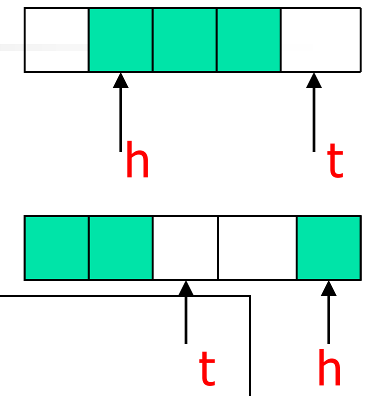
Implementing a Stack

- Using an array
  - Keep an index to the "top" of stack




```
Class Stack {
  int S[0..99];
  int top := -1;
  void push(int x) {
    if (top==99) print "stack overflow";
    else {top++; S[top] := x;}
  }
  int pop() {
    if (top==-1) print "stack underflow";
    else {top--; return S[top+1];}
  }
}
```

Implementing a Queue



- Using an array
  - Use two variables to keep head and tail
  - Wrap around

```
Class Queue {
   int Q[0..99];
   // NB: size 100 but stores only 99 elements
   int h := 0, t := 0;
   void Enqueue(int x) {
      if ((t+1)%100 == h) print "queue full";
      else {Q[t]:=x; t:=(t+1)%100;}
   }
   int Dequeue() {
      if (t == h) print "queue empty";
      else {x:=Q[h]; h:=(h+1)%100; return x;}
   }
}
```

Simple Sorting and Searching

Computers have historically spent more time sorting than doing anything else.

- D. Knuth

Linear Search

- Consider a very simple problem: search for an element x in an array A with n elements
- Trivial algorithm: search one by one

```
// search for input x in array A[1..n]
for i := 1 to n {
  if (A[i]==x) {
    print i;
    return;
  }
}
print "not found";
```

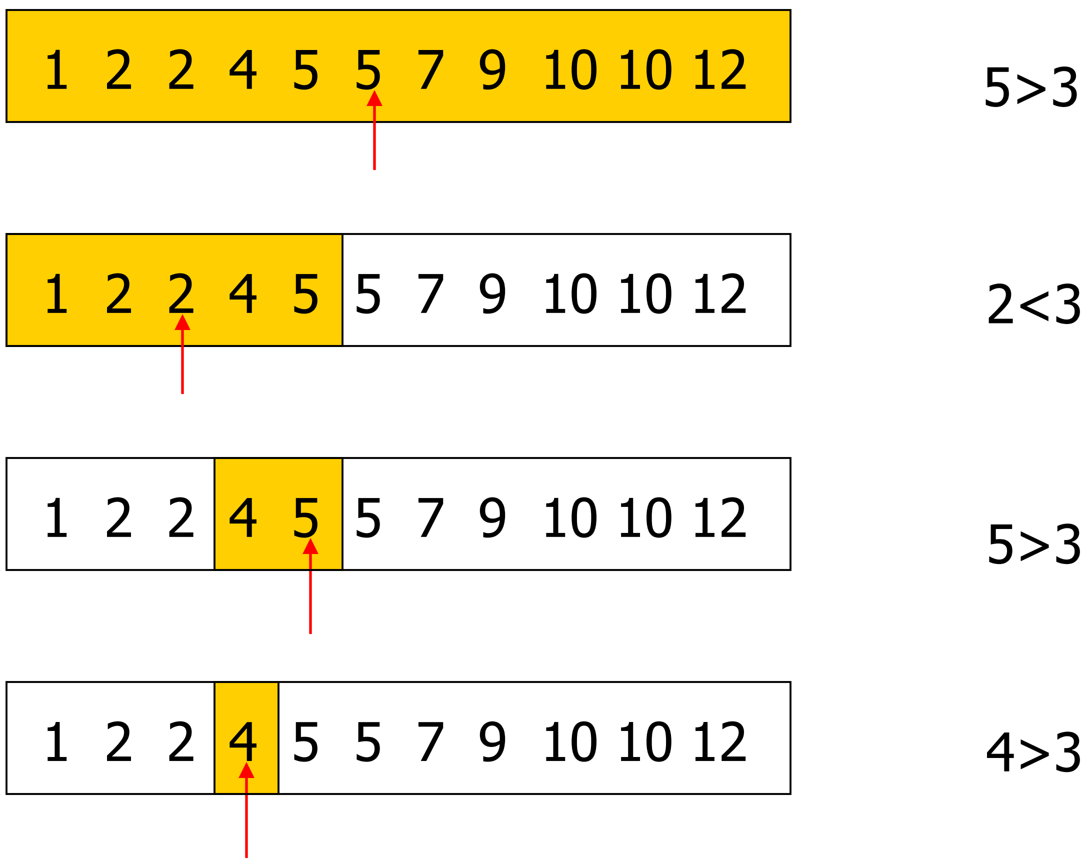
- Time complexity: $O(n)$

Binary Search

- If the elements are already sorted, can we do it faster?
  - (How do you look up a word in a dictionary?)
- *Binary search*
  - Idea: we can reduce the search space by half by checking the middle element
  - Suppose elements sorted in increasing order
  - If middle element > x, x can only be in 1st half
  - If middle element < x, x can only be in 2nd half
  - *Recursively* (or iteratively) handle one of the two halves
    - (because we are now facing the *same* problem of *smaller* size)

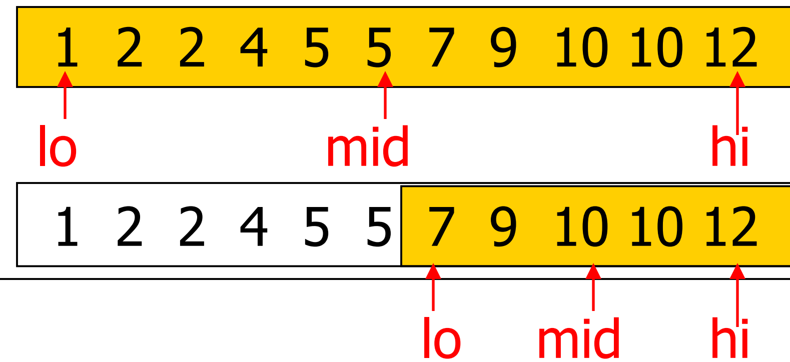
Binary Search: Example

- Search for 3



Binary Search: Non-recursive

- Use two indices lo and hi to indicate the range of the array we are searching




```
Binary-Search(A, n, x)
{
  lo := 1, hi := n
  while (lo <= hi) {
    mid := round((lo+hi)/2)
    if (A[mid]==x) return mid // found
    else if (A[mid]>x) hi:=mid-1 // lower half
    else lo := mid+1 // upper half
  }
}
```

Binary Search: Complexity

- Content inside while loop: O(1) time
- How many times is the while loop executed?
  - Each execution reduces the range (hi – lo + 1) by half
  - hi – lo + 1 is n at the beginning, 1 at the end
  - $n \to n/2 \to n/4 \ldots \to 2 \to 1$: $\log_2 n$ steps
  - Log n executions
- Overall time complexity O(log n)
  - Better than linear search
- See Chapter 3 for recursive version

Sorting

- Problem: given n items with an ordering, arrange them in ascending/descending order
  - E.g. sorting numbers
  - How do you arrange a hand of playing cards?
- A fundamental problem
- This chapter: two simple algorithms
  - Later: advanced algorithms (so, your usual way of arranging playing cards is not optimal…)

  - Try the animations: http://www.sorting-algorithms.com/

Selection Sort

- Idea: find the smallest number, swap it with the first number, and repeat for the remaining numbers
- Example:
  - 3 1 5 8 5 2 7
  - 1 3 5 8 5 2 7
  - 1 2 5 8 5 3 7
  - 1 2 3 8 5 5 7
  - 1 2 3 5 8 5 7
  - 1 2 3 5 5 8 7
  - 1 2 3 5 5 7 8

Selection Sort: Algorithm

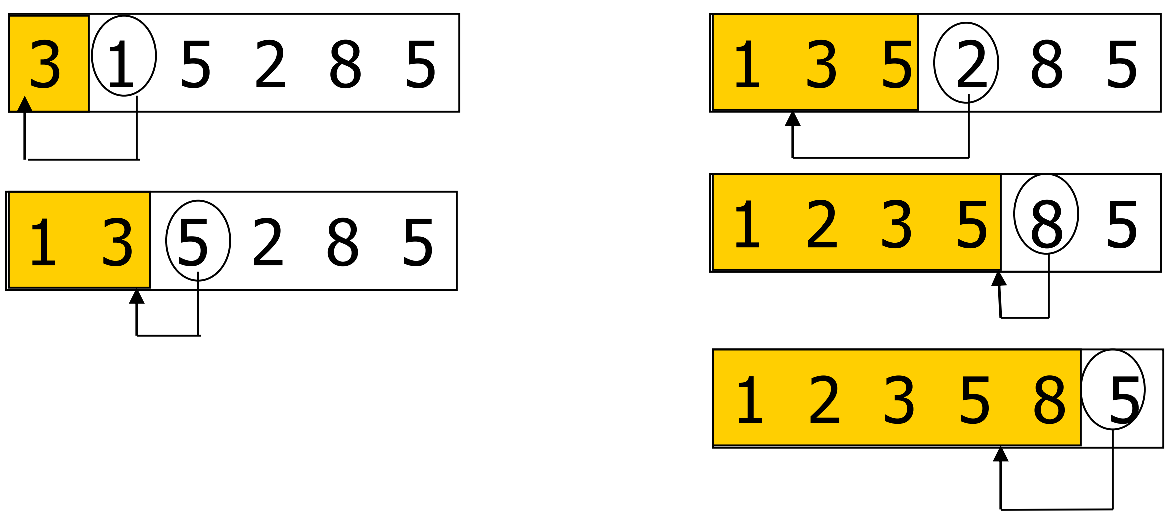
```
// sort array A[1..n]
Selection-Sort(A)
{
  for i := 1 to n-1 {
    // find minimum in A[i..n]
    min := i
    for j := i+1 to n
      if (A[j] < A[min]) min := j
      // now min contains the index of
         minimum position
    swap the values of A[i] and A[min]
  }
}
```

Selection Sort: Complexity

- There are two for loops
  - Outer (for i = …): n-1 times
  - Inner (for j = …): at most n-1 times
  - Within nested loops: O(1) time
- Time complexity: $O(n^2)$

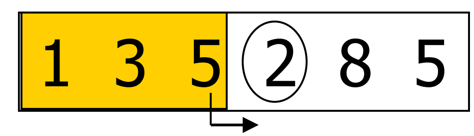
Insertion Sort

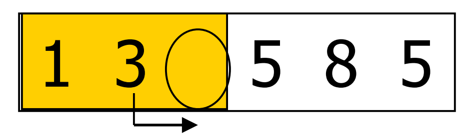
- Idea: maintain that A[1..i] is already sorted.
  - Insert A[i+1] in the correct position within A[1..i]
  - Now, A[1..i+1] is sorted
  - Repeat the procedure to further extend until the whole array is sorted
- Example:

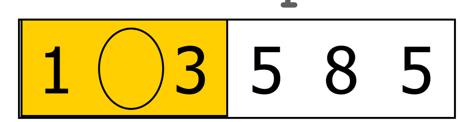


Insertion Sort: Algorithm

```
// sort array A[1..n]
Insertion-Sort(A)
{
  for i := 2 to n {
    j := i-1
    x := A[i]
    while (j>0 and x<A[j]) {
      A[j+1] := A[j]
      j := j-1                    // Move backwards
    }
    A[j+1] := x   // j+1 is correct insert point
  }
}
```







Insertion Sort: Complexity

- For loop (element A[i] to be processed): n times
  - While loop (find correct position): at most n times
- $O(n^2)$ again
  - Later, we will know there are actually faster algorithms!

Binary Insertion Sort

- Do we really need O(n) time to find correct position to insert?
  - The subarray A[1..i] is already sorted
  - Binary search for the correct position!
- Binary insertion sort
  - Number of comparisons: O(n log n)
- However: we need to move the elements after insertion! (which takes O(n) time)
  - Hence, time still $O(n^2)$

Selection vs. Insertion Sort

- Selection Sort:
  - Once a position is found for an element, it remains there
- Insertion Sort:
  - Benefits from partially-sorted input
  - Can be done on-line