Outline of these notes

- Review of basic data structures
- Searching in a sorted array/binary search: the algorithm, analysis, proof of optimality
- Sorting, part 1: insertion sort, selection sort

Basic Data structures

Prerequisite material. Review [GT Chapters 2–4, 6] as necessary)

- Arrays, dynamic arrays
- Linked lists
- Stacks, queues
- Dictionaries, hash tables
- Binary trees

Arrays, Dynamic arrays, Linked lists

Arrays, Dynamic arrays, Linked lists

- Arrays:

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:
  - Similar to arrays, but size can be increased or decreased

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:
  - Similar to arrays, but size can be increased or decreased
  - `ArrayList` in Java, `list` in Python

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:
  - Similar to arrays, but size can be increased or decreased
  - `ArrayList` in Java, `list` in Python
- Linked lists:

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:
  - Similar to arrays, but size can be increased or decreased
  - `ArrayList` in Java, `list` in Python
- Linked lists:
  - Collection of nodes that form a linear ordering.

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:
  - Similar to arrays, but size can be increased or decreased
  - `ArrayList` in Java, `list` in Python
- Linked lists:
  - Collection of nodes that form a linear ordering.
    - The list has a first node and a last node

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:
  - Similar to arrays, but size can be increased or decreased
  - `ArrayList` in Java, `list` in Python
- Linked lists:
  - Collection of nodes that form a linear ordering.
    - The list has a first node and a last node
    - Each node has a next node and a previous node (possibly null)

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:
  - Similar to arrays, but size can be increased or decreased
  - `ArrayList` in Java, `list` in Python
- Linked lists:
  - Collection of nodes that form a linear ordering.
    - The list has a first node and a last node
    - Each node has a next node and a previous node (possibly null)
  - Inserting or deleting an item in the middle of linked list is fast.

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:
  - Similar to arrays, but size can be increased or decreased
  - `ArrayList` in Java, `list` in Python
- Linked lists:
  - Collection of nodes that form a linear ordering.
    - The list has a first node and a last node
    - Each node has a next node and a previous node (possibly null)
  - Inserting or deleting an item in the middle of linked list is fast.
  - Accessing a cell given its index (i.e., finding the kth item in the list) is slow.

Stacks and Queues

Stacks and Queues

- Stacks:

Stacks and Queues

- Stacks:
  - Container of objects that are inserted and removed according to Last-In First-Out (LIFO) principle:

Stacks and Queues

- Stacks:
  - Container of objects that are inserted and removed according to Last-In First-Out (LIFO) principle:
    - Only the most-recently inserted object can be removed.

Stacks and Queues

- Stacks:
  - Container of objects that are inserted and removed according to Last-In First-Out (LIFO) principle:
    - Only the most-recently inserted object can be removed.
  - Insert and remove are usually called `push` and `pop`

Stacks and Queues

- Stacks:
  - Container of objects that are inserted and removed according to Last-In First-Out (LIFO) principle:
    - Only the most-recently inserted object can be removed.
  - Insert and remove are usually called `push` and `pop`
- Queues (often called FIFO Queues)

Stacks and Queues

- Stacks:
  - Container of objects that are inserted and removed according to Last-In First-Out (LIFO) principle:
    - Only the most-recently inserted object can be removed.
  - Insert and remove are usually called `push` and `pop`
- Queues (often called FIFO Queues)
  - Container of objects that are inserted and removed according to First-In First-Out (FIFO) principle:

Stacks and Queues

- Stacks:
  - Container of objects that are inserted and removed according to Last-In First-Out (LIFO) principle:
    - Only the most-recently inserted object can be removed.
  - Insert and remove are usually called `push` and `pop`
- Queues (often called FIFO Queues)
  - Container of objects that are inserted and removed according to First-In First-Out (FIFO) principle:
    - Only the element that has been in the queue the longes can be removed.

Stacks and Queues

- Stacks:
  - Container of objects that are inserted and removed according to Last-In First-Out (LIFO) principle:
    - Only the most-recently inserted object can be removed.
  - Insert and remove are usually called `push` and `pop`
- Queues (often called FIFO Queues)
  - Container of objects that are inserted and removed according to First-In First-Out (FIFO) principle:
    - Only the element that has been in the queue the longes can be removed.
  - Insert and remove are usually called `enqueue` and `dequeue`

Stacks and Queues

- Stacks:
  - Container of objects that are inserted and removed according to Last-In First-Out (LIFO) principle:
    - Only the most-recently inserted object can be removed.
  - Insert and remove are usually called `push` and `pop`
- Queues (often called FIFO Queues)
  - Container of objects that are inserted and removed according to First-In First-Out (FIFO) principle:
    - Only the element that has been in the queue the longes can be removed.
  - Insert and remove are usually called `enqueue` and `dequeue`
  - Elements are inserted at the rear of the queue and are removed from the front

Dictionaries/Maps

Dictionaries/Maps

- Dictionaries

Dictionaries/Maps

- Dictionaries
  - A Dictionary (or Map) stores `<key,value>` pairs, which are often referred to as items

Dictionaries/Maps

- Dictionaries
  - A Dictionary (or Map) stores `<key,value>` pairs, which are often referred to as items
  - There can be at most item with a given key.

Dictionaries/Maps

- Dictionaries
  - A Dictionary (or Map) stores `<key,value>` pairs, which are often referred to as items
  - There can be at most item with a given key.
  - Examples:

Dictionaries/Maps

- Dictionaries
  - A Dictionary (or Map) stores `<key,value>` pairs, which are often referred to as items
  - There can be at most item with a given key.
  - Examples:
    1. `<Student ID, Student data>`

Dictionaries/Maps

- Dictionaries
  - A Dictionary (or Map) stores `<key,value>` pairs, which are often referred to as items
  - There can be at most item with a given key.
  - Examples:
    1. `<Student ID, Student data>`
    2. `<Object ID, Object data>`

Hashing

Hashing

An efficient method for implementing a dictionary.

Hashing

An efficient method for implementing a dictionary. Uses

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.Commonly used collision strategies are:

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.Commonly used collision strategies are:
  - Chaining

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.Commonly used collision strategies are:
  - Chaining
  - Open addressing: linear probing, quadratic probing, double hashing

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.Commonly used collision strategies are:
  - Chaining
  - Open addressing: linear probing, quadratic probing, double hashing
  - Cuckoo hashing

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.Commonly used collision strategies are:
  - Chaining
  - Open addressing: linear probing, quadratic probing, double hashing
  - Cuckoo hashing

Hashing is fast:

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.Commonly used collision strategies are:
  - Chaining
  - Open addressing: linear probing, quadratic probing, double hashing
  - Cuckoo hashing

Hashing is fast:

- $O(1)$ expected time for access, insertion

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.Commonly used collision strategies are:
  - Chaining
  - Open addressing: linear probing, quadratic probing, double hashing
  - Cuckoo hashing

Hashing is fast:

- $O(1)$ expected time for access, insertion
- Cuckoo hashing improves the access time to $O(1)$ worst-case time. Insertion time remains $O(1)$ expected time.

Hashing

An efficient method for implementing a dictionary. Uses

- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.Commonly used collision strategies are:
  - Chaining
  - Open addressing: linear probing, quadratic probing, double hashing
  - Cuckoo hashing

Hashing is fast:

- $O(1)$ expected time for access, insertion
- Cuckoo hashing improves the access time to $O(1)$ worst-case time. Insertion time remains $O(1)$ expected time.

Disadvantages on next slide.

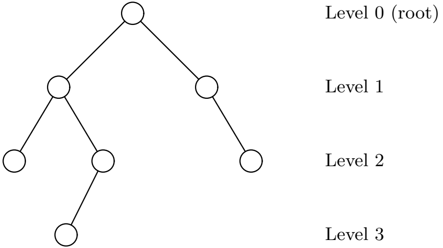
Binary Trees: a quick review

Binary Trees: a quick review

We will use as a data structure and as a tool for analyzing algorithms.

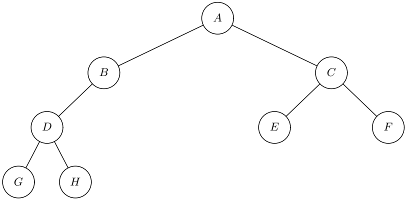
Binary Trees: a quick review

We will use as a data structure and as a tool for analyzing algorithms.

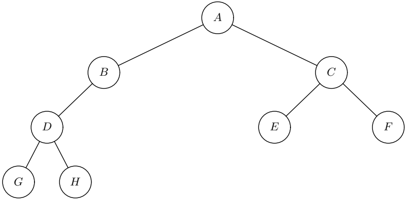


The depth of a binary tree is the maximum of the levels of all its leaves.

Traversing binary trees

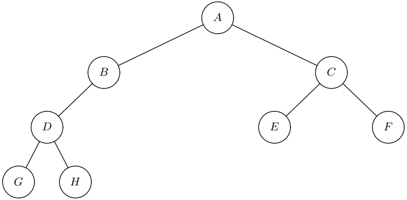


Traversing binary trees



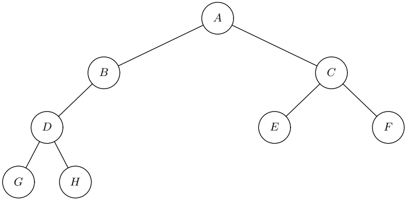
- Preorder: root, left subtree (in preorder), right subtree (in preorder): *ABDGHCEF*

Traversing binary trees



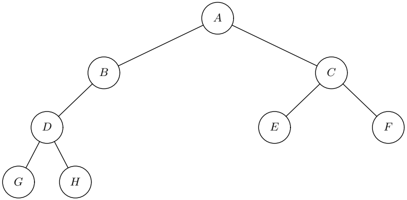
- Preorder: root, left subtree (in preorder), right subtree (in preorder): *ABDGHCEF*
- Inorder: left subtree (in inorder), root, right subtree (in inorder): *GDHBAECF*

Traversing binary trees



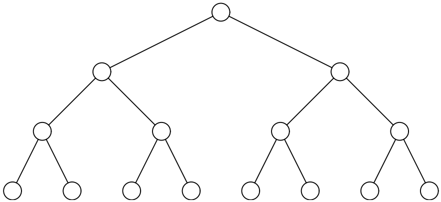
- Preorder: root, left subtree (in preorder), right subtree (in preorder): *ABDGHCEF*
- Inorder: left subtree (in inorder), root, right subtree (in inorder): *GDHBAECF*
- Postorder: left subtree (in postorder), right subtree (in postorder), root: *GHDBEFCA*

Traversing binary trees

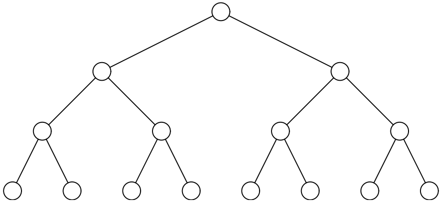


- Preorder: root, left subtree (in preorder), right subtree (in preorder): *ABDGHCEF*
- Inorder: left subtree (in inorder), root, right subtree (in inorder): *GDHBAECF*
- Postorder: left subtree (in postorder), right subtree (in postorder), root: *GHDBEFCA*
- Breadth-first order (level order): level 0 left-to-right, then level 1 left-to-right, . . . : *ABCDEFGH*

Facts about binary trees

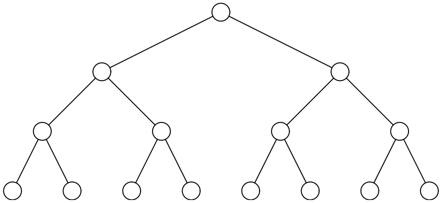


Facts about binary trees



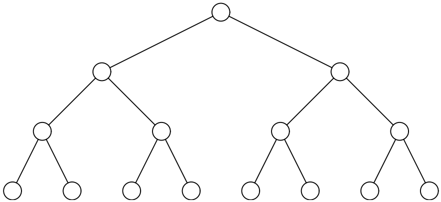
1. There are at most 2^k nodes at level k.

Facts about binary trees



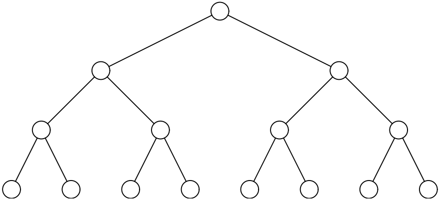
1. There are at most 2^k nodes at level k.
2. A binary tree with depth d has:
   - At most 2^d leaves.

Facts about binary trees



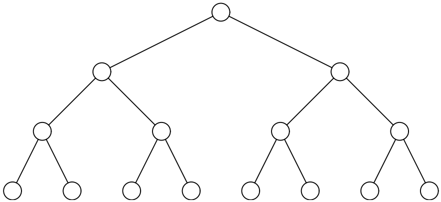
1. There are at most 2^k nodes at level k.
2. A binary tree with depth d has:
   - At most 2^d leaves.
   - At most $2^{d+1} - 1$ nodes.

Facts about binary trees



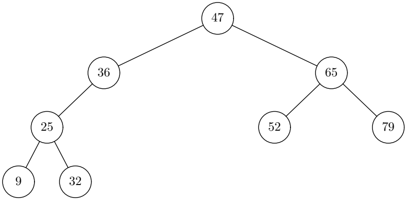
1. There are at most 2^k nodes at level k.
2. A binary tree with depth d has:
   - At most 2^d leaves.
   - At most $2^{d+1} - 1$ nodes.
3. A binary tree with n leaves has depth $\geq \lceil \lg n \rceil$.

Facts about binary trees

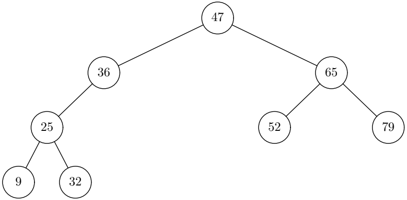


1. There are at most 2^k nodes at level k.
2. A binary tree with depth d has:
   - At most 2^d leaves.
   - At most $2^{d+1} - 1$ nodes.
3. A binary tree with n leaves has depth $\geq \lceil \lg n \rceil$.
4. A binary tree with n nodes has depth $\geq \lfloor \lg n \rfloor$.

Binary search trees

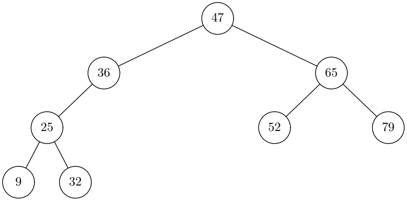


Binary search trees



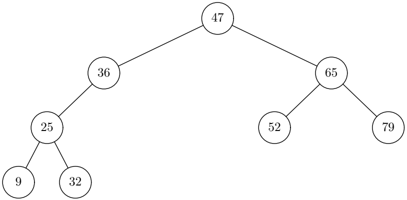
- Function as ordered dictionaries. (Can find successors, predecessors)

Binary search trees



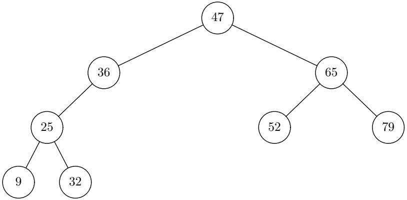
- Function as ordered dictionaries. (Can find successors, predecessors)
- find, insert, and remove can all be done in $O(h)$ time ($h =$ tree height)

Binary search trees



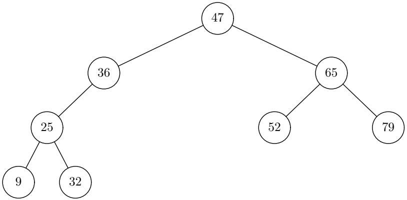
- Function as ordered dictionaries. (Can find successors, predecessors)
- find, insert, and remove can all be done in $O(h)$ time ($h =$ tree height)
- AVL trees, Red-Black Trees, Weak AVL trees: $h = O(\log n)$, so find, insert, and remove can all be done in $O(\log n)$ time.

Binary search trees



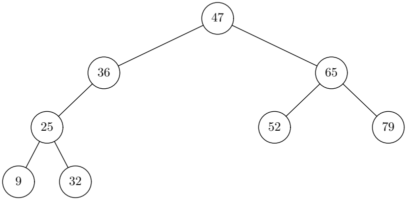
- Function as ordered dictionaries. (Can find successors, predecessors)
- find, insert, and remove can all be done in $O(h)$ time ($h =$ tree height)
- AVL trees, Red-Black Trees, Weak AVL trees: $h = O(\log n)$, so find, insert, and remove can all be done in $O(\log n)$ time.
- Splay trees and Skip Lists: alternatives to balanced trees

Binary search trees



- Function as ordered dictionaries. (Can find successors, predecessors)
- find, insert, and remove can all be done in $O(h)$ time ($h =$ tree height)
- AVL trees, Red-Black Trees, Weak AVL trees: $h = O(\log n)$, so find, insert, and remove can all be done in $O(\log n)$ time.
- Splay trees and Skip Lists: alternatives to balanced trees
- Can traverse the tree and list all items in $O(n)$ time.

Binary search trees



- Function as ordered dictionaries. (Can find successors, predecessors)
- find, insert, and remove can all be done in $O(h)$ time ($h =$ tree height)
- AVL trees, Red-Black Trees, Weak AVL trees: $h = O(\log n)$, so find, insert, and remove can all be done in $O(\log n)$ time.
- Splay trees and Skip Lists: alternatives to balanced trees
- Can traverse the tree and list all items in $O(n)$ time.
- [GT] Chapters 3–4 for details

Binary Search: Searching in a sorted array

Binary Search: Searching in a sorted array

- Input is a sorted array A and an item x.

Binary Search: Searching in a sorted array

- Input is a sorted array A and an item x.
- Problem is to locate x in the array.

Binary Search: Searching in a sorted array

- Input is a sorted array A and an item x.
- Problem is to locate x in the array.
- Several variants of the problem, for example

Binary Search: Searching in a sorted array

- Input is a sorted array A and an item x.
- Problem is to locate x in the array.
- Several variants of the problem, for example. . .
  1. Determine whether x is stored in the array

Binary Search: Searching in a sorted array

- Input is a sorted array A and an item x.
- Problem is to locate x in the array.
- Several variants of the problem, for example...
  1. Determine whether x is stored in the array
  2. Find the largest i such that $A[i] \leq x$ (with a reasonable convention if $x < A[0]$).

Binary Search: Searching in a sorted array

- Input is a sorted array A and an item x.
- Problem is to locate x in the array.
- Several variants of the problem, for example...
  1. Determine whether x is stored in the array
  2. Find the largest i such that $A[i] \leq x$ (with a reasonable convention if $x < A[0]$).

  We will focus on the first variant.

Binary Search: Searching in a sorted array

- Input is a sorted array A and an item x.
- Problem is to locate x in the array.
- Several variants of the problem, for example...
  1. Determine whether x is stored in the array
  2. Find the largest i such that $A[i] \leq x$ (with a reasonable convention if $x < A[0]$).

  We will focus on the first variant.
- We will show that binary search is an optimal algorithm for solving this problem.

Binary Search: Searching in a sorted array

Binary Search: Searching in a sorted array

Input:	A:	Sorted array with n entries $[0..n-1]$
	x:	Item we are seeking

Binary Search: Searching in a sorted array

Input: A: Sorted array with n entries $[0..n-1]$
x: Item we are seeking

Output: Location of x, if x found
-1, if x not found

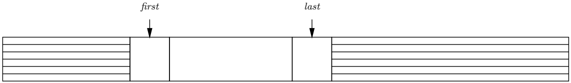
Binary Search: Searching in a sorted array

Input: A: Sorted array with n entries $[0..n-1]$
x: Item we are seeking

Output: Location of x, if x found
-1, if x not found

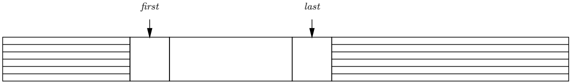
```
def binarySearch(A,x,first,last)
if first > last:
  return (-1)
else:
  mid = ⌊(first+last)/2⌋
  if x == A[mid]:
    return mid
  else if x < A[mid]:
    return binarySearch(A,x,first,mid-1)
  else:
    return binarySearch(A,x,mid+1,last)

binarySearch(A,x,0,n-1)
```

Correctness of Binary Search



Correctness of Binary Search

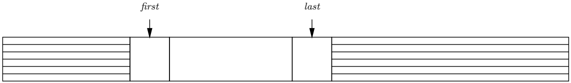
We need to prove two things:



Correctness of Binary Search

We need to prove two things:

1. If x is in the array, its location in the array (its index) is between *first* and *last*, inclusive.

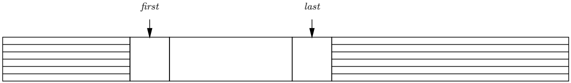


Correctness of Binary Search

We need to prove two things:

1. If x is in the array, its location in the array (its index) is between *first* and *last*, inclusive.
   Note that this is equivalent to:

   Either x is not in the array, or its location is between first and last, inclusive.



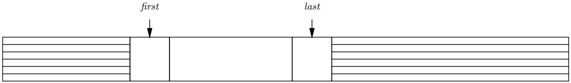
Correctness of Binary Search

We need to prove two things:

1. If x is in the array, its location in the array (its index) is between *first* and *last*, inclusive.
   Note that this is equivalent to:

   > *Either x is not in the array, or its location is between first and last, inclusive.*

2. On each recursive call, the difference *last* − *first* gets strictly smaller.



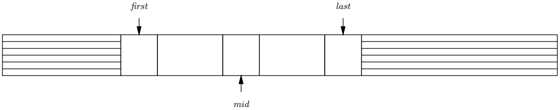
Correctness of Binary Search

To prove that the invariant continues to hold, we need to consider three cases.

Correctness of Binary Search

To prove that the invariant continues to hold, we need to consider three cases.

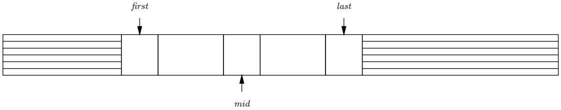
1. $last \geq first + 2$



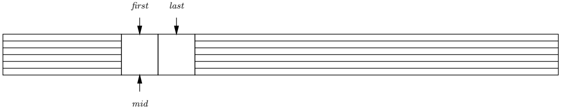
Correctness of Binary Search

To prove that the invariant continues to hold, we need to consider three cases.

1. *last* $\geq$ *first* $+ 2$



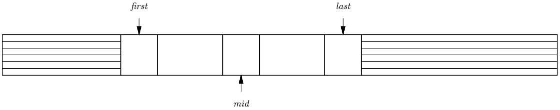
2. *last* $=$ *first* $+ 1$



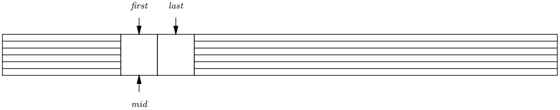
Correctness of Binary Search

To prove that the invariant continues to hold, we need to consider three cases.

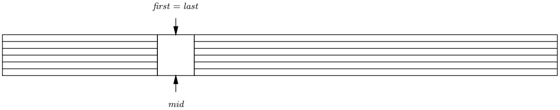
1. *last* $\geq$ *first* $+ 2$



2. *last* $=$ *first* $+ 1$



3. *last* $=$ *first*



Binary Search: Analysis of Running Time

Binary Search: Analysis of Running Time

- We will count the number of 3-way comparisons of x against elements of A. (also known as decisions)

Binary Search: Analysis of Running Time

- We will count the number of 3-way comparisons of x against elements of A. (also known as decisions)
- Rationale:

Binary Search: Analysis of Running Time

- We will count the number of 3-way comparisons of x against elements of A. (also known as decisions)
- Rationale:
  1. This is the essentially the same as the number of recursive calls. Every recursive call, except for possibly the very last one, results in a 3-way comparison.

Binary Search: Analysis of Running Time

- We will count the number of 3-way comparisons of x against elements of A. (also known as decisions)
- Rationale:
  1. This is the essentially the same as the number of recursive calls. Every recursive call, except for possibly the very last one, results in a 3-way comparison.
  2. Gives us a way to compare binary search against other algorithms that solve the same problem: searching for an item in an array by comparing the item against array entries.

Binary Search: Analysis of Running Time (continued)

Binary Search: Analysis of Running Time (continued)

- Binary search in an array of size 1: 1 decision

Binary Search: Analysis of Running Time (continued)

- Binary search in an array of size 1: 1 decision
- Binary search in an array of size $n > 1$: after 1 decision, either we are done, or the problem is reduced to binary search in a subarray with a worst-case size of $\lfloor n/2 \rfloor$

Binary Search: Analysis of Running Time (continued)

- Binary search in an array of size 1: 1 decision
- Binary search in an array of size $n > 1$: after 1 decision, either we are done, or the problem is reduced to binary search in a subarray with a worst-case size of $\lfloor n/2 \rfloor$
- So the worst-case time to do binary search on an array of size n is $T(n)$, where $T(n)$ satisfies the equation

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{otherwise} \end{cases}$$

Binary Search: Analysis of Running Time (continued)

- Binary search in an array of size 1: 1 decision
- Binary search in an array of size $n > 1$: after 1 decision, either we are done, or the problem is reduced to binary search in a subarray with a worst-case size of $\lfloor n/2 \rfloor$
- So the worst-case time to do binary search on an array of size n is $T(n)$, where $T(n)$ satisfies the equation

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{otherwise} \end{cases}$$

- The solution to this equation is:

$$T(n) = \lfloor \lg n \rfloor + 1$$

This can be proved by induction.

Binary Search: Analysis of Running Time (continued)

- Binary search in an array of size 1: 1 decision
- Binary search in an array of size $n > 1$: after 1 decision, either we are done, or the problem is reduced to binary search in a subarray with a worst-case size of $\lfloor n/2 \rfloor$
- So the worst-case time to do binary search on an array of size n is $T(n)$, where $T(n)$ satisfies the equation

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{otherwise} \end{cases}$$

- The solution to this equation is:

$$T(n) = \lfloor \lg n \rfloor + 1$$

  This can be proved by induction.
- So binary search does $\lfloor \lg n \rfloor + 1$ 3-way comparisons on an array of size n, in the worst case.

Optimality of binary search

Optimality of binary search

- We will establish a lower bound on the worst-case number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.

Optimality of binary search

- We will establish a lower bound on the worst-case number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.
- The lower bound we will establish is $\lfloor \lg n \rfloor + 1$ 3-way comparisons.

Optimality of binary search

- We will establish a lower bound on the worst-case number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.
- The lower bound we will establish is $\lfloor \lg n \rfloor + 1$ 3-way comparisons.
- Since Binary Search performs within this bound, it is optimal.

Optimality of binary search

- We will establish a lower bound on the worst-case number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.
- The lower bound we will establish is $\lfloor \lg n \rfloor + 1$ 3-way comparisons.
- Since Binary Search performs within this bound, it is optimal.
- Our lower bound is established using a Decision Tree model.

Optimality of binary search

- We will establish a lower bound on the worst-case number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.
- The lower bound we will establish is $\lfloor \lg n \rfloor + 1$ 3-way comparisons.
- Since Binary Search performs within this bound, it is optimal.
- Our lower bound is established using a Decision Tree model.
- Note that the bound is exact (not just asymptotic)

Optimality of binary search

- We will establish a lower bound on the worst-case number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.
- The lower bound we will establish is $\lfloor \lg n \rfloor + 1$ 3-way comparisons.
- Since Binary Search performs within this bound, it is optimal.
- Our lower bound is established using a Decision Tree model.
- Note that the bound is exact (not just asymptotic)
- Our lower bound is on the worst case

Optimality of binary search

- We will establish a lower bound on the worst-case number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.
- The lower bound we will establish is $\lfloor \lg n \rfloor + 1$ 3-way comparisons.
- Since Binary Search performs within this bound, it is optimal.
- Our lower bound is established using a Decision Tree model.
- Note that the bound is exact (not just asymptotic)
- Our lower bound is on the worst case
  - It says: for every algorithm for finding an item in an array of size n, there is some input that forces it to perform $\lfloor \lg n \rfloor + 1$ comparisons.

Optimality of binary search

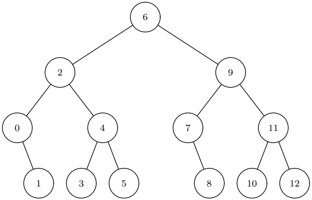
- We will establish a lower bound on the worst-case number of decisions required to find an item in an array, using only 3-way comparisons of the item against array entries.
- The lower bound we will establish is $\lfloor \lg n \rfloor + 1$ 3-way comparisons.
- Since Binary Search performs within this bound, it is optimal.
- Our lower bound is established using a Decision Tree model.
- Note that the bound is exact (not just asymptotic)
- Our lower bound is on the worst case
  - It says: for every algorithm for finding an item in an array of size n, there is some input that forces it to perform $\lfloor \lg n \rfloor + 1$ comparisons.
  - It does not say: for every algorithm for finding an item in an array of size n, every input forces it to perform $\lfloor \lg n \rfloor + 1$ comparisons.

The decision tree model for searching in an array

The decision tree model for searching in an array

Consider any algorithm that searches for an item x in an array A of size n by comparing entries in A against x. Any such algorithm can be modeled as a decision tree:

Example: Decision tree for binary search with $n = 13$:

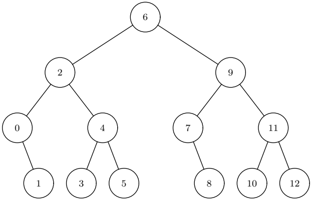


The decision tree model for searching in an array

Consider any algorithm that searches for an item x in an array A of size n by comparing entries in A against x. Any such algorithm can be modeled as a decision tree:

- Each node is labeled with an integer $\in \{0 \ldots n-1\}$.

Example: Decision tree for binary search with $n = 13$:

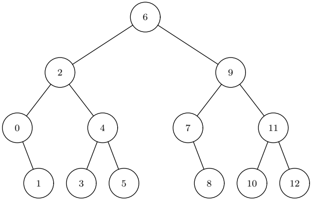


The decision tree model for searching in an array

Consider any algorithm that searches for an item x in an array A of size n by comparing entries in A against x. Any such algorithm can be modeled as a decision tree:

- Each node is labeled with an integer $\in \{0 \ldots n-1\}$.
- A node labeled i represents a 3-way comparison between x and $A[i]$.

Example: Decision tree for binary search with $n = 13$:

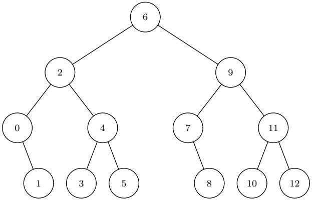


The decision tree model for searching in an array

Consider any algorithm that searches for an item x in an array A of size n by comparing entries in A against x. Any such algorithm can be modeled as a decision tree:

- Each node is labeled with an integer $\in \{0 \ldots n-1\}$.
- A node labeled i represents a 3-way comparison between x and $A[i]$.
- The left subtree of a node labeled i describes the decision tree for what happens if $x < A[i]$.

Example: Decision tree for binary search with $n = 13$:

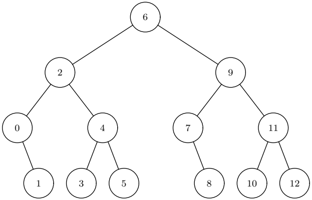


The decision tree model for searching in an array

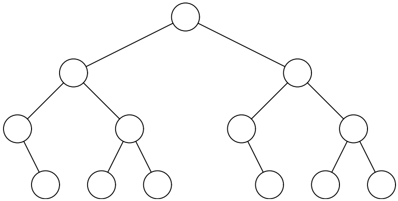
Consider any algorithm that searches for an item x in an array A of size n by comparing entries in A against x. Any such algorithm can be modeled as a decision tree:

- Each node is labeled with an integer $\in \{0 \ldots n-1\}$.
- A node labeled i represents a 3-way comparison between x and $A[i]$.
- The left subtree of a node labeled i describes the decision tree for what happens if $x < A[i]$.
- The right subtree of a node labeled i describes the decision tree for what happens if $x > A[i]$.

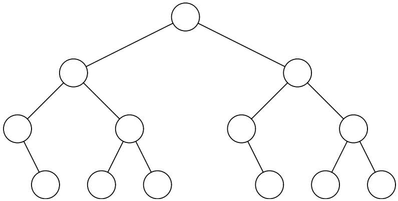
Example: Decision tree for binary search with $n = 13$:



Lower bound on locating an item in an array of size n

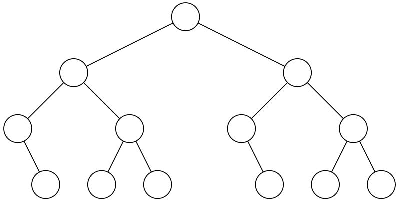


Lower bound on locating an item in an array of size n



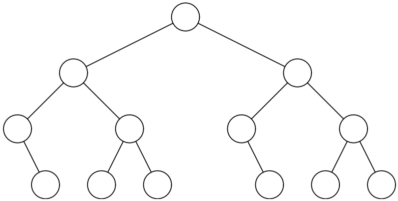
1. Any algorithm for searching an array of size n can be modeled by a decision tree with at least n nodes.

Lower bound on locating an item in an array of size n



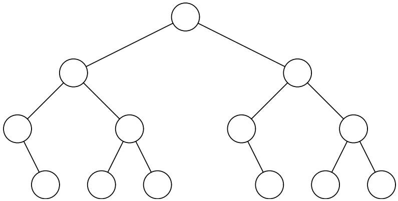
1. Any algorithm for searching an array of size n can be modeled by a decision tree with at least n nodes.
2. Since the decision tree is a binary tree with n nodes, the depth is at least $\lfloor \lg n \rfloor$.

Lower bound on locating an item in an array of size n



1. Any algorithm for searching an array of size n can be modeled by a decision tree with at least n nodes.
2. Since the decision tree is a binary tree with n nodes, the depth is at least $\lfloor \lg n \rfloor$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree $+1$. (Remember, root has depth 0).

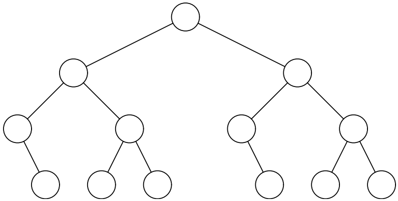
Lower bound on locating an item in an array of size n



1. Any algorithm for searching an array of size n can be modeled by a decision tree with at least n nodes.
2. Since the decision tree is a binary tree with n nodes, the depth is at least $\lfloor \lg n \rfloor$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree $+1$. (Remember, root has depth 0).

Hence any algorithm for locating an item in an array of size n using only comparisons must perform at least $\lfloor \lg n \rfloor + 1$ comparisons in the worst case.

Lower bound on locating an item in an array of size n



1. Any algorithm for searching an array of size n can be modeled by a decision tree with at least n nodes.
2. Since the decision tree is a binary tree with n nodes, the depth is at least $\lfloor \lg n \rfloor$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree $+1$. (Remember, root has depth 0).

Hence any algorithm for locating an item in an array of size n using only comparisons must perform at least $\lfloor \lg n \rfloor + 1$ comparisons in the worst case.

So binary search is optimal with respect to worst-case performance.

Sorting

Sorting

- Rearranging a list of items in nondescending order.

Sorting

- Rearranging a list of items in nondescending order.
- Useful preprocessing step (e.g., for binary search)

Sorting

- Rearranging a list of items in nondescending order.
- Useful preprocessing step (e.g., for binary search)
- Important step in other algorithms

Sorting

- Rearranging a list of items in nondescending order.
- Useful preprocessing step (e.g., for binary search)
- Important step in other algorithms
- Illustrates more general algorithmic techniques

Sorting

- Rearranging a list of items in nondescending order.
- Useful preprocessing step (e.g., for binary search)
- Important step in other algorithms
- Illustrates more general algorithmic techniques

We will discuss

Sorting

- Rearranging a list of items in nondescending order.
- Useful preprocessing step (e.g., for binary search)
- Important step in other algorithms
- Illustrates more general algorithmic techniques

We will discuss

- Comparison-based sorting algorithms (Insertion sort, Selection Sort, Quicksort, Mergesort, Heapsort)

Sorting

- Rearranging a list of items in nondescending order.
- Useful preprocessing step (e.g., for binary search)
- Important step in other algorithms
- Illustrates more general algorithmic techniques

We will discuss

- Comparison-based sorting algorithms (Insertion sort, Selection Sort, Quicksort, Mergesort, Heapsort)
- Bucket-based sorting methods

Sorting

- Rearranging a list of items in nondescending order.
- Useful preprocessing step (e.g., for binary search)
- Important step in other algorithms
- Illustrates more general algorithmic techniques

We will discuss in the class

- Comparison-based sorting algorithms (Insertion sort, Selection Sort, Quicksort, Mergesort, Heapsort)
- Bucket-based sorting methods

Comparison-based sorting

- Basic operation: compare two items.

Comparison-based sorting

- Basic operation: compare two items.
- Abstract model.

Comparison-based sorting

- Basic operation: compare two items.
- Abstract model.
- Advantage: doesn't use specific properties of the data items. So same algorithm can be used for sorting integers, strings,

Comparison-based sorting

- Basic operation: compare two items.
- Abstract model.
- Advantage: doesn't use specific properties of the data items. So same algorithm can be used for sorting integers, strings, etc.
- Disadvantage: under certain circumstances, specific properties of the data item can speed up the sorting process.

Comparison-based sorting

- Basic operation: compare two items.
- Abstract model.
- Advantage: doesn't use specific properties of the data items. So same algorithm can be used for sorting integers, strings, etc.
- Disadvantage: under certain circumstances, specific properties of the data item can speed up the sorting process.
- Measure of time: number of comparisons

Comparison-based sorting

- Basic operation: compare two items.
- Abstract model.
- Advantage: doesn't use specific properties of the data items. So same algorithm can be used for sorting integers, strings, etc.
- Disadvantage: under certain circumstances, specific properties of the data item can speed up the sorting process.
- Measure of time: number of comparisons
  - Consistent with philosophy of counting basic operations, discussed earlier.

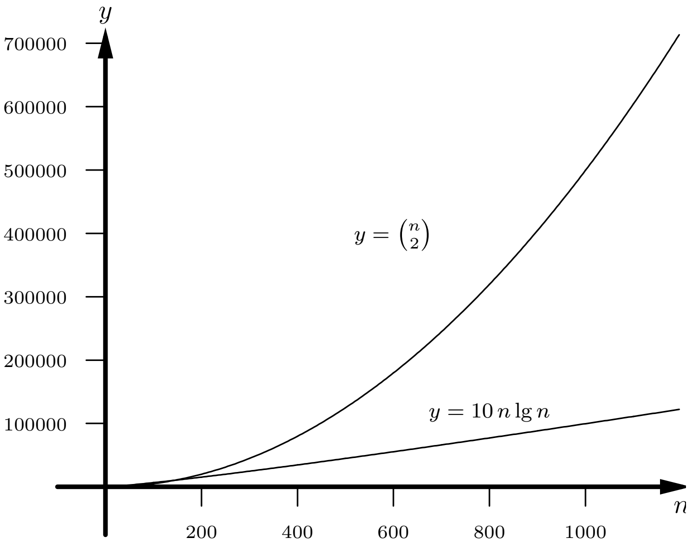
Comparison-based sorting

- Basic operation: compare two items.
- Abstract model.
- Advantage: doesn't use specific properties of the data items. So same algorithm can be used for sorting integers, strings, etc.
- Disadvantage: under certain circumstances, specific properties of the data item can speed up the sorting process.
- Measure of time: number of comparisons
  - Consistent with philosophy of counting basic operations, discussed earlier.
  - Misleading if other operations dominate (e.g., if we sort by moving items around without comparing them)

Comparison-based sorting

- Basic operation: compare two items.
- Abstract model.
- Advantage: doesn't use specific properties of the data items. So same algorithm can be used for sorting integers, strings, etc.
- Disadvantage: under certain circumstances, specific properties of the data item can speed up the sorting process.
- Measure of time: number of comparisons
  - Consistent with philosophy of counting basic operations, discussed earlier.
  - Misleading if other operations dominate (e.g., if we sort by moving items around without comparing them)
- Comparison-based sorting has lower bound of $\Omega(n \log n)$ comparisons. (We will prove this.)

$\Theta(n \log n)$ work vs. quadratic ($\Theta(n^2)$) work



Some terminology

Some terminology

- A permutation of a sequence of items is a reordering of the sequence. A sequence of n items has $n!$ distinct permutations.

Some terminology

- A permutation of a sequence of items is a reordering of the sequence. A sequence of n items has $n!$ distinct permutations.
- Note: Sorting is the problem of finding a particular distinguished permutation of a list.

Some terminology

- A permutation of a sequence of items is a reordering of the sequence. A sequence of n items has $n!$ distinct permutations.
- Note: Sorting is the problem of finding a particular distinguished permutation of a list.
- An inversion in a sequence or list is a pair of items such that the larger one precedes the smaller one.

Some terminology

- A permutation of a sequence of items is a reordering of the sequence. A sequence of n items has $n!$ distinct permutations.
- Note: Sorting is the problem of finding a particular distinguished permutation of a list.
- An inversion in a sequence or list is a pair of items such that the larger one precedes the smaller one.

Example: The list

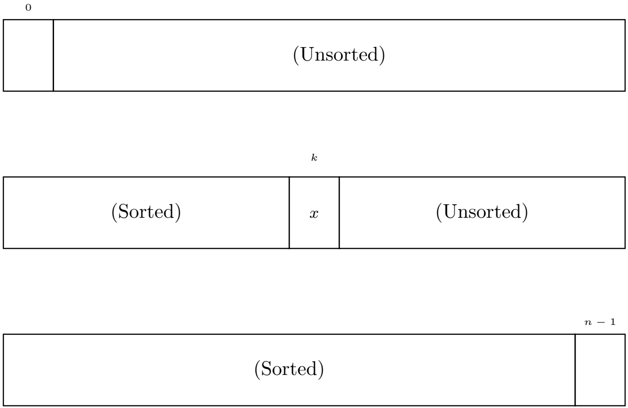
18 29 12 15 32 10

has 9 inversions:

$$\{(18,12), (18,15), (18,10), (29,12), (29,15), (29,10), (12,10), (15,10), (32,10)\}$$

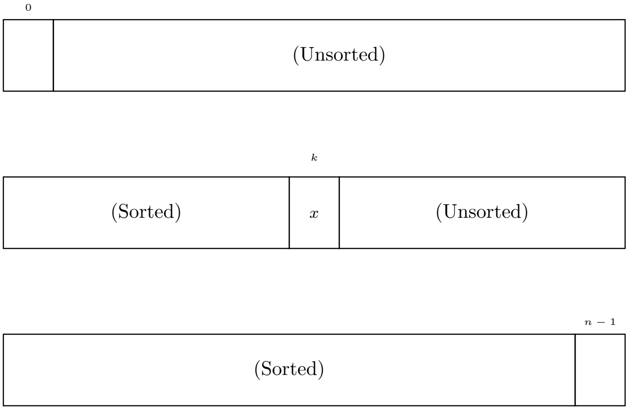
Insertion sort

Insertion sort



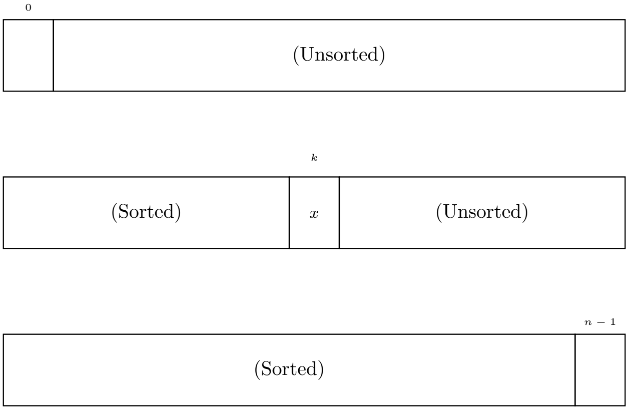
Insertion sort

- Work from left to right across array

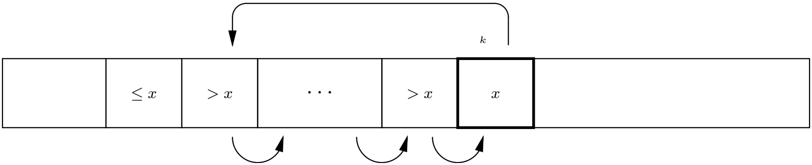


Insertion sort

- Work from left to right across array
- Insert each item in correct position with respect to (sorted) elements to its left



Insertion sort pseudocode




```
def insertionSort(n, A):
    for k = 1 to n-1:
        x = A[k]
        j = k-1
        while (j >= 0) and (A[j] > x):
            A[j+1] = A[j]
            j = j-1
        A[j+1] = x
```

Insertion sort example

23	19	42	17	85	38

23	19	42	17	85	38

19	23	42	17	85	38

19	23	42	17	85	38

17	19	23	42	85	38

17	19	23	42	85	38

17	19	23	38	42	85

Analysis of Insertion Sort

- Worst-case running time:

Analysis of Insertion Sort

- Worst-case running time:
  - On kth iteration of outer loop, element $A[k]$ is compared with at most k elements:
    $A[k-1]$, $A[k-2]$, ..., $A[0]$.

Analysis of Insertion Sort

- Worst-case running time:
  - On kth iteration of outer loop, element $A[k]$ is compared with at most k elements:
    $A[k-1], A[k-2], \ldots, A[0]$.
  - Total number comparisons over all iterations is at most:

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2).$$

Analysis of Insertion Sort

- Worst-case running time:
  - On kth iteration of outer loop, element $A[k]$ is compared with at most k elements:
    $A[k-1]$, $A[k-2]$, ..., $A[0]$.
  - Total number comparisons over all iterations is at most:

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2).$$

  - Insertion Sort is a bad choice when n is large. ($O(n^2)$ vs. $O(n \log n)$).

Analysis of Insertion Sort

- Worst-case running time:
  - On kth iteration of outer loop, element $A[k]$ is compared with at most k elements:
    $A[k-1], A[k-2], \ldots, A[0]$.
  - Total number comparisons over all iterations is at most:

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2).$$

  - Insertion Sort is a bad choice when n is large. ($O(n^2)$ vs. $O(n \log n)$).
  - Insertion Sort is a good choice when n is small. (Constant hidden in the "big oh" is small).

Analysis of Insertion Sort

- Worst-case running time:
  - On kth iteration of outer loop, element $A[k]$ is compared with at most k elements:
    $A[k-1], A[k-2], \ldots, A[0]$.
  - Total number comparisons over all iterations is at most:

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2).$$

  - Insertion Sort is a bad choice when n is large. ($O(n^2)$ vs. $O(n \log n)$).
  - Insertion Sort is a good choice when n is small. (Constant hidden in the "big oh" is small).
  - Insertion Sort is efficient if the input is "almost sorted":

$$\text{Time} \leq n - 1 + (\#\ \text{inversions})$$

Analysis of Insertion Sort

- Worst-case running time:
  - On kth iteration of outer loop, element $A[k]$ is compared with at most k elements:
    $A[k-1], A[k-2], \ldots, A[0]$.
  - Total number comparisons over all iterations is at most:

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2).$$

  - Insertion Sort is a bad choice when n is large. ($O(n^2)$ vs. $O(n \log n)$).
  - Insertion Sort is a good choice when n is small. (Constant hidden in the "big oh" is small).
  - Insertion Sort is efficient if the input is "almost sorted":

$$\text{Time} \leq n - 1 + (\#\ \text{inversions})$$

- Storage: in place: $O(1)$ extra storage

Selection Sort

Selection Sort

- Two variants:

Selection Sort

- Two variants:
  1. Repeatedly (for i from 0 to $n-1$) find the minimum value, output it, delete it.
     - Values are output in sorted order

Selection Sort

- Two variants:
  1. Repeatedly (for i from 0 to $n-1$) find the minimum value, output it, delete it.
     - Values are output in sorted order
  2. Repeatedly (for i from $n-1$ down to 1)

Selection Sort

- Two variants:
  1. Repeatedly (for i from 0 to $n-1$) find the minimum value, output it, delete it.
     - Values are output in sorted order
  2. Repeatedly (for i from $n-1$ down to 1)
     - Find the maximum of $A[0]$,$A[1]$,...,$A[i]$.

Selection Sort

- Two variants:
  1. Repeatedly (for i from 0 to $n-1$) find the minimum value, output it, delete it.
     - Values are output in sorted order
  2. Repeatedly (for i from $n-1$ down to 1)
     - Find the maximum of $A[0]$,$A[1]$,...,$A[i]$.
     - Swap this value with $A[i]$ (no-op if it is already $A[i]$).

Selection Sort

- Two variants:
  1. Repeatedly (for i from 0 to $n-1$) find the minimum value, output it, delete it.
     - Values are output in sorted order
  2. Repeatedly (for i from $n-1$ down to 1)
     - Find the maximum of $A[0]$,$A[1]$,. . . ,$A[i]$.
     - Swap this value with $A[i]$ (no-op if it is already $A[i]$).
- Both variants run in $O(n^2)$ time if we use the straightforward approach to finding the maximum/minimum.

Selection Sort

- Two variants:
  1. Repeatedly (for i from 0 to $n-1$) find the minimum value, output it, delete it.
     - Values are output in sorted order
  2. Repeatedly (for i from $n-1$ down to 1)
     - Find the maximum of $A[0]$,$A[1]$,...,$A[i]$.
     - Swap this value with $A[i]$ (no-op if it is already $A[i]$).
- Both variants run in $O(n^2)$ time if we use the straightforward approach to finding the maximum/minimum.