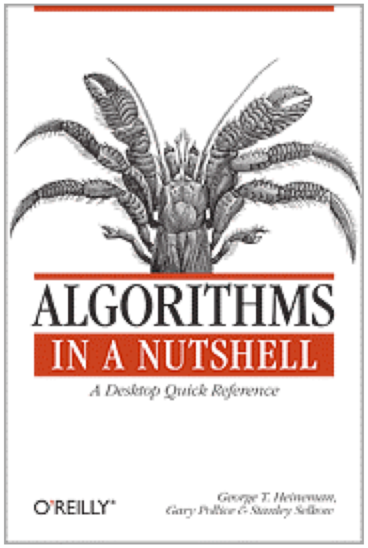
Algorithms in a Nutshell



Session 3

Searching

10:40 – 11:20

Outline

- Searching Principles
- Themes
  - Divide and Conquer
  - Space vs. Time
  - Rich Data Structures
- Algorithms
  - BINARY SEARCH, TREE-BASED, HASH-BASED
- Concerns
  - Hash functions, Storage overhead

Searching Principles

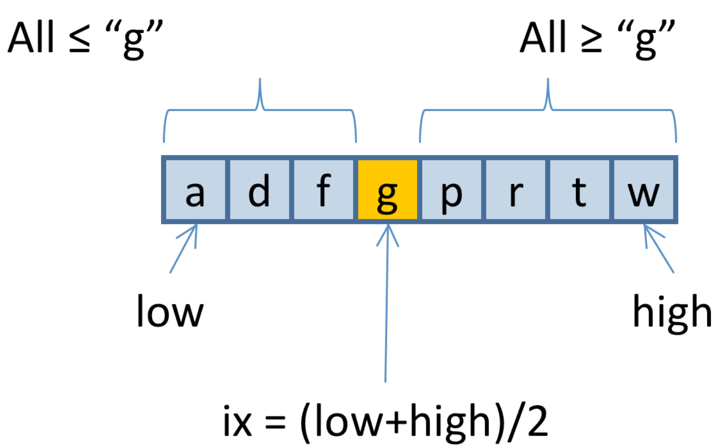
- Given a collection *C* of elements
- Existence
  - Does *C* contain a target element *t*
- Retrieval
  - Return element in *C* that matches target element *t*
- Associative lookup
  - Return information in *C* associated with target key *k*

Unordered Representation

- Must scan each element of C
  - Performance O(n)
- Ordered representations are essential
  - Phone Book
  - Dictionary
  - Aisles at Home Depot

BINARY SEARCH

- Represent C using a sorted array of elements
- Apply divide and conquer
  - Fastest search algorithm for contiguous array
  - Difficult to code without defects!



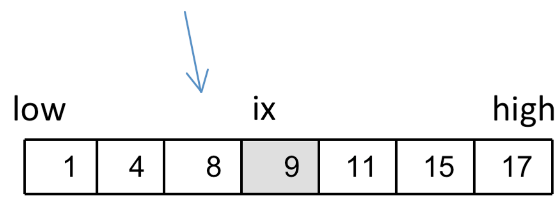
- If "r" is in C it must be in upper half of the array since "r" ≥ "g"
- Each iteration cuts size of array by about half
  - log(n) iterations

112

BINARY SEARCH

```
search(A, t)
    low = 0
    high = n–1
    while (low ≤ high) do
      ix = (low + high)/2
      if (t = A[ix]) then
        return true
      else if (t < A[ix]) then
        high = ix–1
      else low  = ix + 1
    return false
end
```

Search (A, 11)



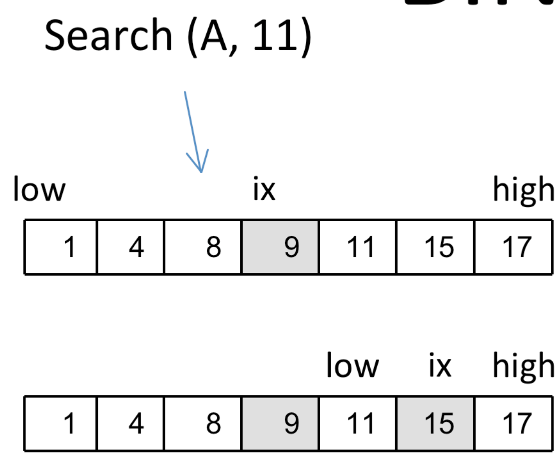
Best case	Average case	Worst case
O (1)	O (log n)	O (log n)

- Implementation
  - Tight **while** loop
  - Integer arithmetic for $\lfloor(low+high)/2\rfloor$
  - Returns **true** when found
- Comparison function
  - Three value logic: <, =, >
  - Avoid multiple comparisons

112

BINARY SEARCH

```
search(A, t)
  low = 0
  high = n–1
  while (low ≤ high) do
    ix = (low + high)/2
    if (t = A[ix]) then
      return true
    else if (t < A[ix]) then
      high = ix–1
    else low  = ix + 1
  return false
end
```

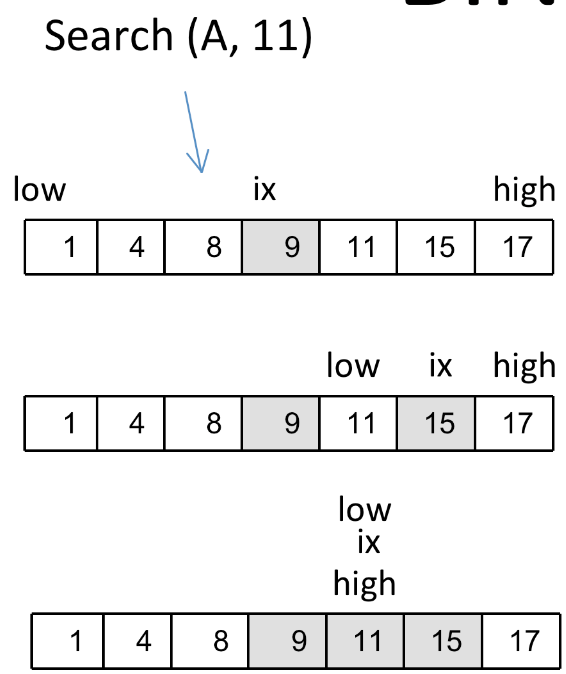


Best case	Average case	Worst case
O (1)	O (log n)	O (log n)

- Implementation
  - Tight **while** loop
  - Integer arithmetic for $\lfloor (low+high)/2 \rfloor$
  - Returns **true** when found
- Comparison function
  - Three value logic: <, =, >
  - Avoid multiple comparisons

BINARY SEARCH

```
search(A, t)
  low = 0
  high = n–1
  while (low ≤ high) do
    ix = (low + high)/2
    if (t = A[ix]) then
      return true
    else if (t < A[ix]) then
      high = ix–1
    else low  = ix + 1
  return false
end
```



Best case	Average case	Worst case
O (1)	O (log n)	O (log n)

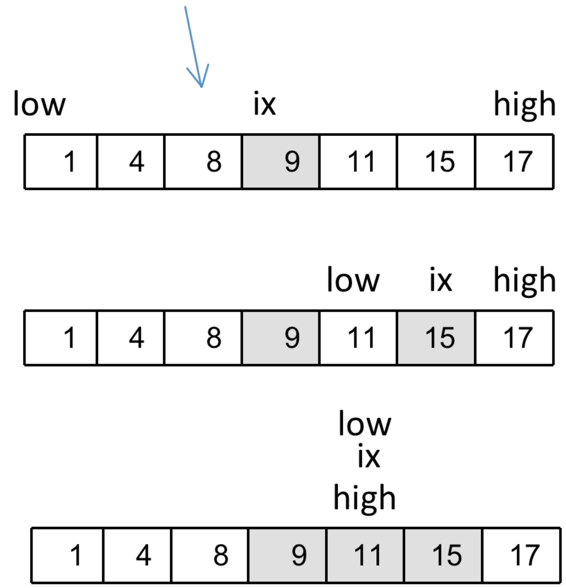
- Implementation
  - Tight **while** loop
  - Integer arithmetic for $\lfloor (low+high)/2 \rfloor$
  - Returns **true** when found
- Comparison function
  - Three value logic: <, =, >
  - Avoid multiple comparisons

112

BINARY SEARCH

```
search(A, t)
    low = 0
    high = n–1
    while (low ≤ high) do
      ix = (low + high)/2
      if (t = A[ix]) then
        return true
      else if (t < A[ix]) then
        high = ix–1
      else low  = ix + 1
    return false
end
```

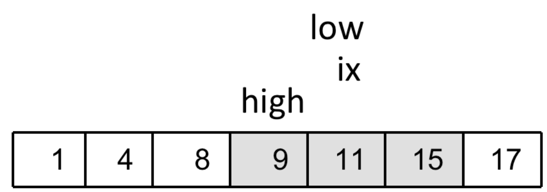
Search (A, 11)



Best case	Average case	Worst case
O (1)	O (log n)	O (log n)

- Implementation
  - Tight **while** loop
  - Integer arithmetic for $\lfloor(\text{low+high})/2\rfloor$
  - Returns **true** when found
- Comparison function
  - Three value logic: <, =, >
  - Avoid multiple comparisons

What if Search (A, 10)?



BINARY SEARCH

- Implementation
  - Tight **while** loop
  - Integer arithmetic for ⌊(low+high)/2⌋
  - Returns **true** when found

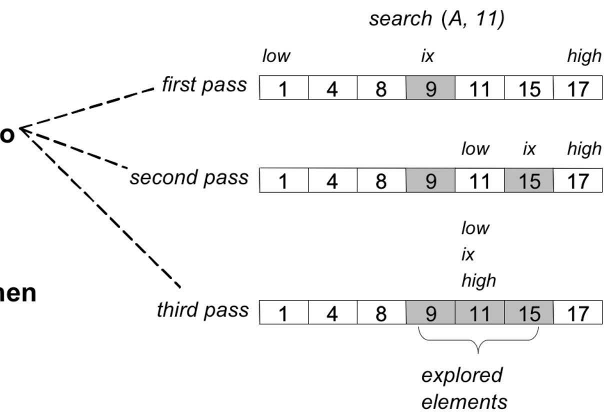
BINARY SEARCH

Best case	Average case	Worst case
O (1)	O (log n)	O (log n)

Array

Divide and Conquer

```
search (A, t)
    low = 0
    high = n–1
    while (low ≤ high) do
      ix = (low + high)/2
      if (t = A[ix]) then
          return true
      else if (t < A[ix]) then
          high = ix– 1
      else low = ix+1
    return false
end
```



- Comparison function
  - Three value logic: <, =, >
  - Avoid multiple comparisons

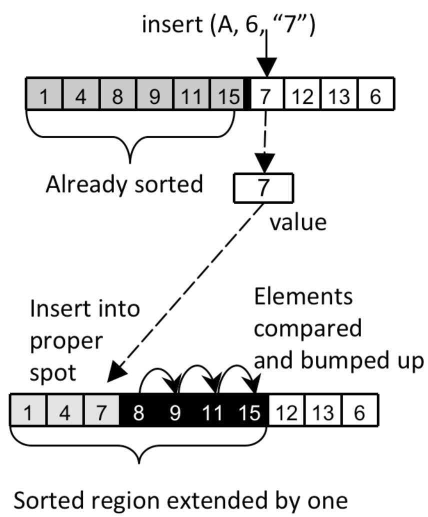
Code Check

- Show actual running code
  - Handout
  - Debug example

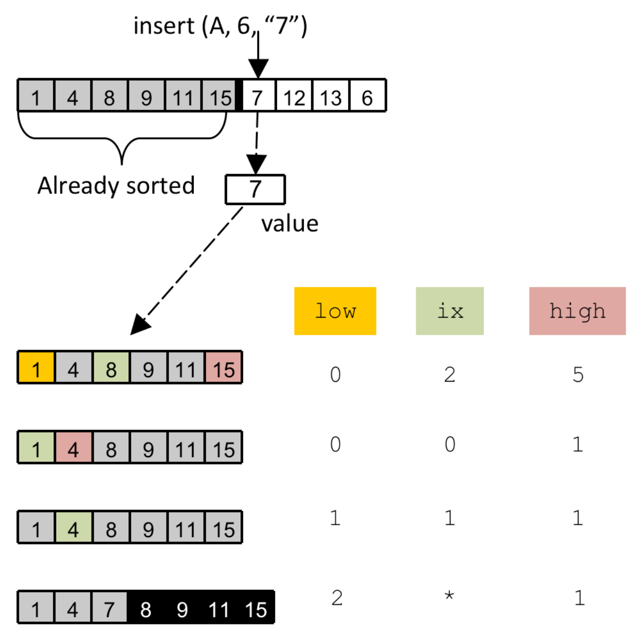
BINARY INSERTION SORT

- Use BINARY SEARCH during INSERTION SORT?

INSERTION SORT: 5 Comparisons



BINARY INSERTION SORT: 3 Comparisons



Code Check

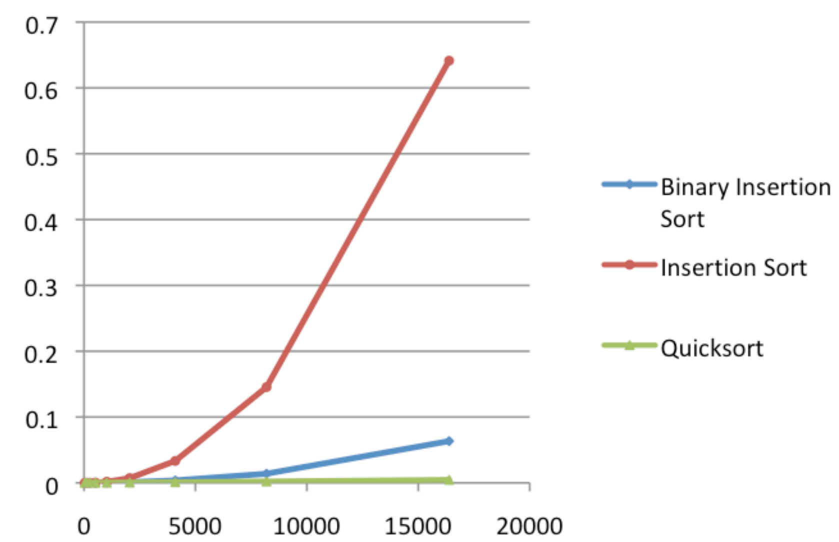
- Locate proper spot
  - BINARY SEARCH
- Make room
  - Bulk move

```c
void sortPointers (char **ar, int n) {
  for (int j = 1; j < n; j++) {
    /** Search for desired target within array */
    int low = 0, high = j-1, ix, rc, sz;
    char *target = ar[j];

    while (low <= high) {
      ix = (low + high)/2;
      rc = strcmp(target, ar[ix]);

      if (rc < 0) {
        /* target is less than ar[i] */
        high = ix - 1;
      } else if (rc > 0) {
        /* target is greater than ar[i] */
        low = ix + 1;
      } else {
        /* found the item. */
        break;
      }
    }

   /** only move if not already properly in place */
   if (low != j) {
     sz = (j-low)*sizeof(char *);
     memmove (&ar[low+1], &ar[low], sz);
     ar[low] = target;
   }
  }
}
```

Comparisons

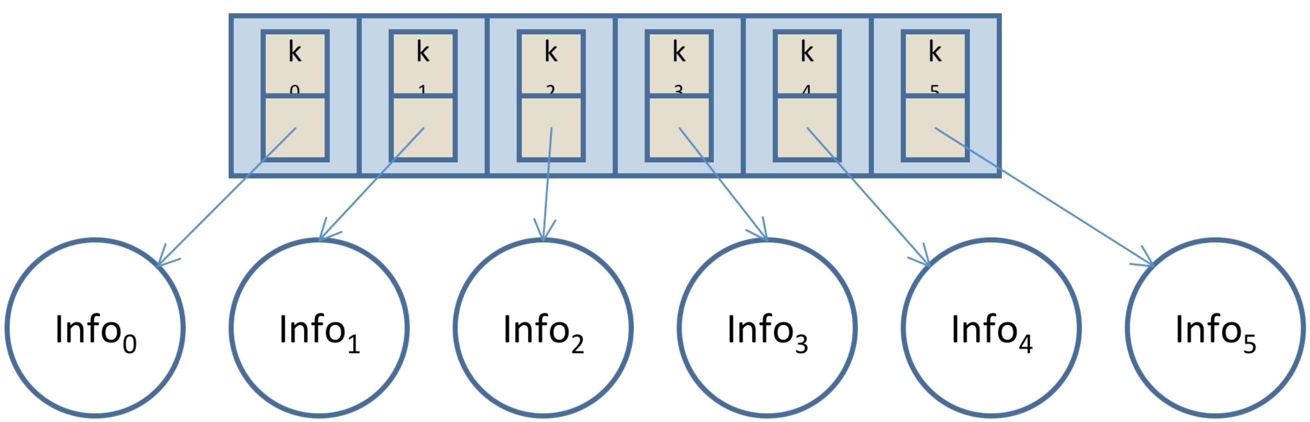
- BINARY INSERTION SORT clear winner over INSERTION SORT
- QUICKSORT still has best performance of three

n	Binary Insertion Sort	Insertion Sort	Quicksort
32	0.000004	0.000003	0.000004
64	0.000007	0.000009	0.000008
128	0.000016	0.00003	0.000016
256	0.000039	0.00011	0.000035
512	0.000104	0.000426	0.000077
1024	0.000315	0.0017	0.000171
2048	0.001	0.0072	0.000389
4096	0.0037	0.0333	0.000897
8192	0.0139	0.1455	0.002
16384	0.0634	0.6414	0.0045



Search types for BINARY SEARCH

- Existence: Does *C* contain element *t*
  – Only need 3-value comparison function
- Associative lookup: Return info associated with key *k*
  – Elements in array store reference to associated info



Code Check

- Code check of example binary search with arrays
- Implementation a bit awkward
  - Constructing the initial array, for instance

Binary Search Weaknesses

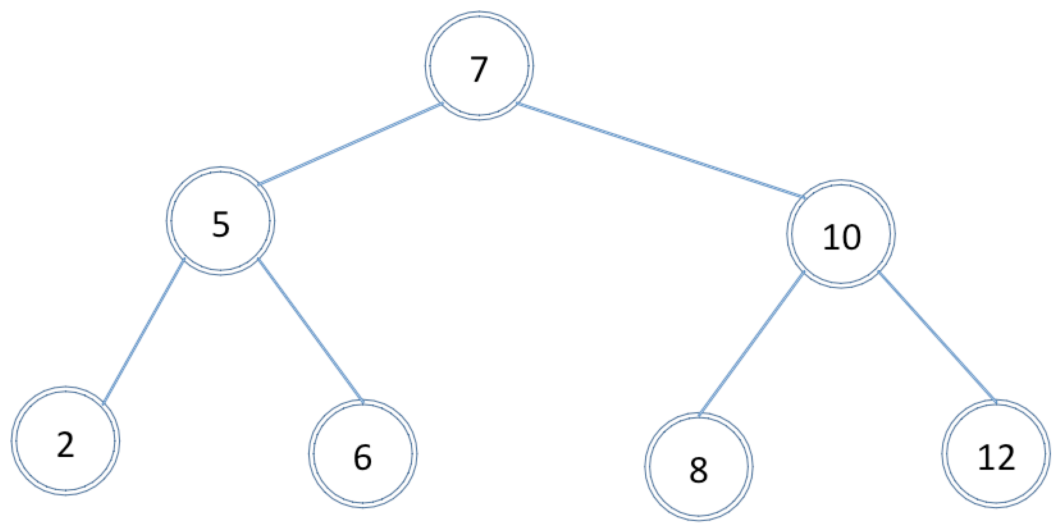
- Costly to support frequent insertion and deletion of elements
- Contiguous storage in an array

- Can an alternate structure address both concerns?
  – Binary Search Tree

Binary Tree Structure

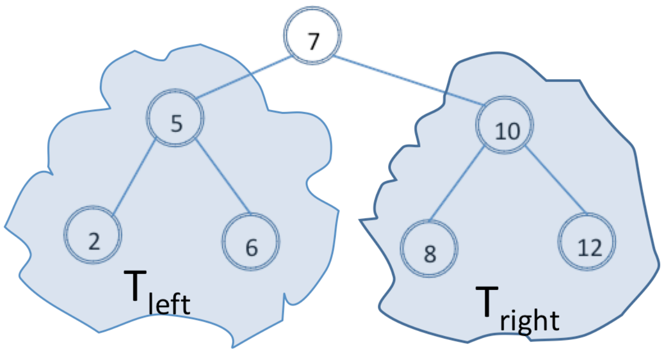
- Recursive data structure
  - Each node may have a *left* and *right* child node
  - Topmost node in the tree is called the *root*

```
class BinaryNode {
   int            value;
   BinaryNode     left;
   BinaryNode     right;
}

class BinaryTree {
   BinaryNode     root;
}
```



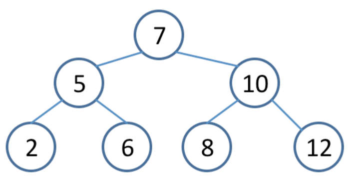
Binary Search Tree Property

- Each node *n* has a key *k*
  - Often the value of the node is simply the key
- Each node *n* refers to two binary search trees
  - T_{left} is tree rooted by left child of *n*
  - T_{right} is tree rooted by right child of *n*
- Keys obey specific ordering
  - All keys in T_{left} for *n* are $\leq k$
  - All keys in T_{right} for *n* are $\geq k$

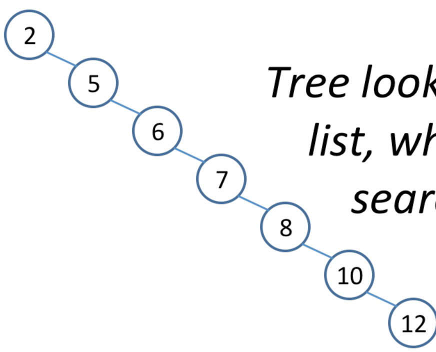


Binary Search Tree Issues

- Reasons to use Binary Search Tree
  - Input data size is unknown
  - Input data is highly dynamic, with significant number of insertions and deletions
- Problems that may arise
  - When a Binary Search Tree is constructed and modified, it may become *unbalanced*



Tree is fully balanced for maximum efficiency



Tree looks more like a linked list, which leads to O(n) search performance

Self-balancing Trees

- Should you choose to use Binary Search Trees
  - Choose a balanced tree structure
- Several choices
  - Red/Black Trees (standard for JDK)
  - AVL Trees (discovered in 1962)
- (Re)balance Tree after insert/delete
  - Insertions and Deletions may unbalance tree

Information Structure for Search

- BINARY SEARCH within sorted array
  - For all valid indices i,j: **if** i ≤ j **then** A[i] ≤ A[j]
- For node *n* (with key *k*) in Binary Search Tree
  - All keys in left sub-tree of *n* are ≤ *k*
  - All keys in right sub-tree of *n* are ≥ *k*
- These structures enforce a global property
  - Can we construct an alternative search structure that provides efficient search? Yes!

HASH-BASED SEARCH

- How to search *C* with *n* elements
  - Break into *b* smaller search problems
- Possible with carefully designed *hash* function
  - Each element $e \in C$ has a key value $k=key(e)$
  - A hash function $h=hash(e)$ uses key value to compute bin $A[h]$ into which to insert *e*

Element	Key	Hash (for table size 7)
hypoplankton	427,589,249	3
unheavenly	427,589,249	3
upheaval	1,440,257,016	2

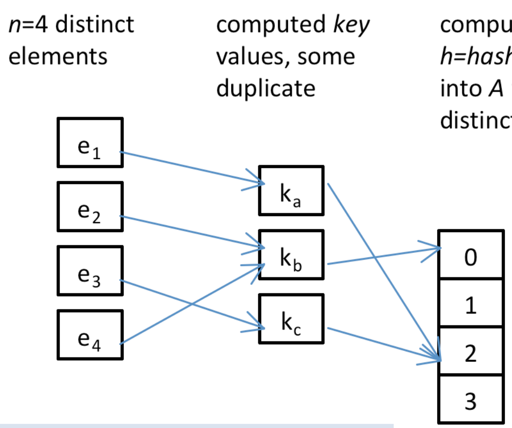
hashCode() is the key

hashCode() % 7 is the hash method

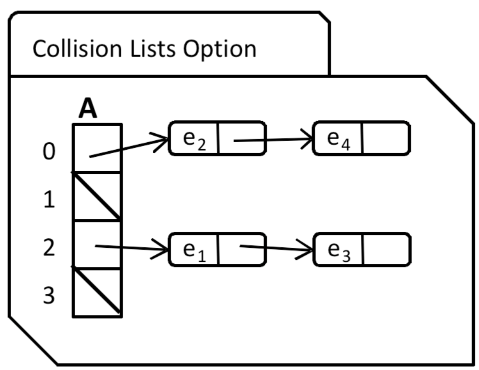
You must know the size of the Hashtable before you can compute hash()

HASH-BASED SEARCH

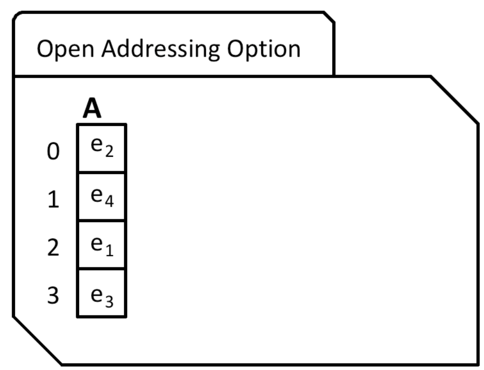
- Collision: Two keys map to same bin $A[h]$
  - Option: Linked lists store elements in each bin
  - Option: Open addressing [see blog entry]



This image is slightly different from Figure 5-4 on page 118



To search for an element, must check each one in the chain for that element's designated bin



*Note that open addressing does impose a global structure. The above structure is **FULL** and no more elements can be added*

Key points for HASH-BASED SEARCH

- Load table: O(n)
  - Construct hash table
- Search table: O(1)
  - With assumptions
  - *hash* function evenly distributed
  - Number of bins "sufficiently large"
- Load factor α
  - Defined as *n/b*

HASH-BASED SEARCH

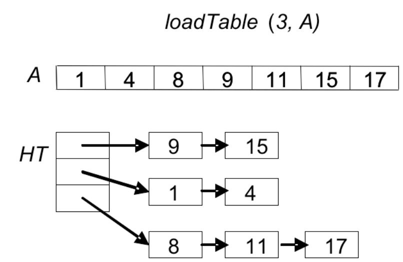
Best case	Average case	Worst case
O (1)	O (1)	O (n)

Array

Hash

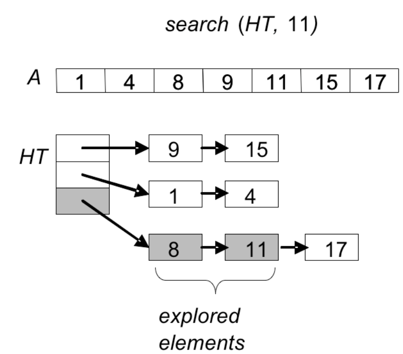
loadTable (size, A)
1. HT = new array of given size
2. **for** i=0 **to** n–1 **do**
3. h = hash(A[i])
4. **if** (HT[h] is empty) **then**
5. HT[h] = new Linked List
6. add A[i] to HT[h]

end



search (HT, t)
1. h = hash (t)
2. list = HT[h]
3. **if** (list is empty) **then**
4. **return false**
5. **if** (list contains t) **then**
6. **return true**
7. **return false**

end



Hashtable Data

- Empirical evaluation: n=213,557
  - What happens with different size b?

b	Load factor α	Min Length	Max Length	Number Unique
4,095	54.04	27	82	0
8,191	27.5	9	46	0
16,383	15	2	28	0
32,767	9.5	0	19	349 (1%)
65,535	6.5	0	13	8,190 (12%)
131,071	5	0	10	41,858 (32%)
262,143	3.5	0	7	94,319 (36%)
524,287	3.5	0	7	142,530 (27%)
1,048,575	2.5	0	5	173,912 (16%)

Weighted: considers only non-empty bins

Hashtable maintenance

- Adding too many objects to a fixed-size hashtable reduces its efficiency
  - Why? Average chain size increases
- Many standard libraries automatically rehash
  - Must be an infrequent operation, since O(n)
  - Can “amortize” costs away over its lifetime
- Java JDK, GNU STL, SGI STL, …
  - Solid implementations. Don’t reinvent the wheel!

Storage Overhead

- BINARY SEARCH
  - No extra memory beyond allocated array
- BINARY TREE SEARCH
  - Left and right pointers: O(n) extra space
- HASH-BASED SEARCH
  - Array of b bins
  - Chained linked lists: O(n) extra space

End notes

- CFP for adding hashing to STL [here]
- STL does not **yet** have hash tables in standard
  - Existing STL implementations do (SGI and GNU)
  - Planned as part of TR1 extension