COMP20003 Algorithms and Data Structures Dictionaries and Data Structures

Nir Lipovetzky
Department of Computing and Information Systems
University of Melbourne
Semester 2

So far…

- We have:
  - Looked at algorithms, fast and slow.
  - Estimated computation time by counting operations.
  - Formalized a system for classifying algorithm efficiency.

Outline of the first few lectures

- Algorithms: general
- This subject: details
- Algorithm efficiency: intuitive
- Computational complexity
- Data structures
  - Basic data structures
  - Algorithms on basic data structures
  - Complexity analysis of algorithms on basic ds's

Textbook

- Skiena: Chapter 3, Data Structures

This section

- A lightning tour of fundamental data structures used for search:
  - Arrays
  - Linked Lists
  - Trees

Abstract Data Types *vs.* Data Structures

- *Abstract* data type: what it does
  - Stack, queue.
  - Dictionary: look up by key.
  - Does not specify an implementation.
- *Concrete* data structure:
  - Array, linked list, tree
  - Can be used to implement abstract data type.

Data structures

- Organizing data is important.
- It is helpful to organize with the task in mind.
- For searching, e.g.:
  - Some high-level languages have inbuilt "dictionaries", or associative arrays (Python, awk).
  - In lower-level languages, dictionaries are implemented directly using a fundamental data structure.

Searching

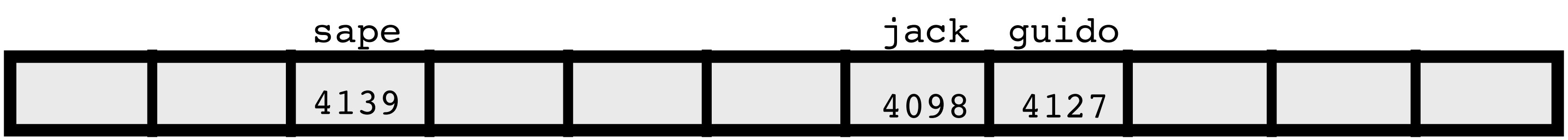
- Search Question:
  - Given a search key,
  - Find the record(s) that correspond to this key.
  - Typically we describe record simplistically with fields key and info (or just key).
- Examples:
  - Students and seat numbers.
  - Telephone books.
- How you organize the data is important.

Dictionaries

- Python built-in dictionary structure.

```
>>> tel = {’jack’: 4098, ’sape’: 4139}
>>> tel[’guido’] = 4127
>>> tel
{’sape’: 4139, ’guido’: 4127, ’jack’: 4098}
>>> tel[’jack’]
4098
```

- The dictionary is implemented using one of the underlying data structures.



Outline of the first few lectures

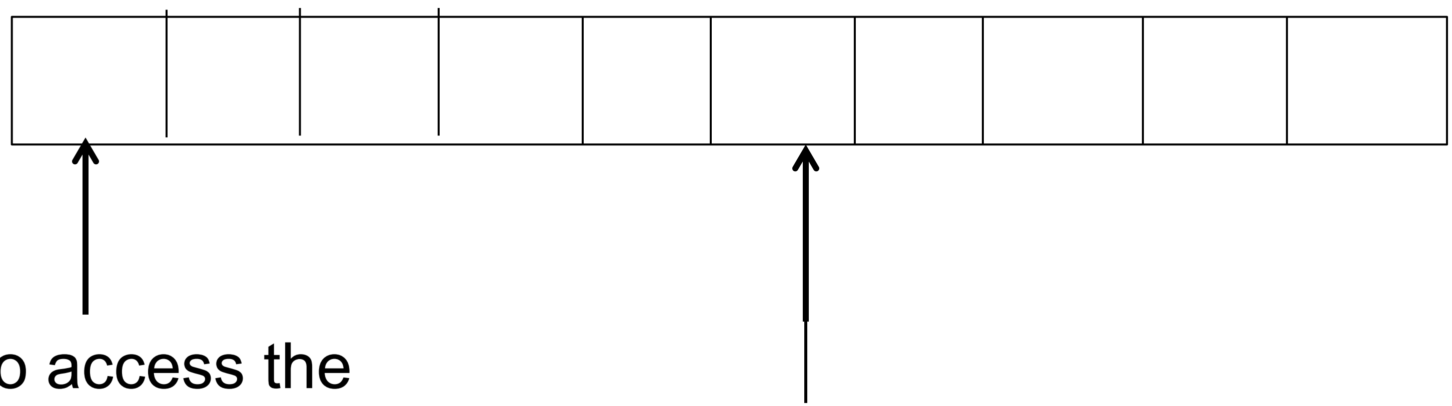
- Algorithms: general
- This subject: details
- Algorithm efficiency: intuitive
- Computational complexity
- Data structures
  - Basic data structures
  - Algorithms on basic data structures
  - Complexity analysis of algorithms on basic ds's

Array

- An array: given an index (location), we can retrieve any item in unit time.
- Mimics the structure of random access memory (RAM), where the index is the memory address.
- If items are in arbitrary order, finding a key in array of size n requires $O(n)$ time.

Arrays in C (lightening review)

```
int  A[10];
```



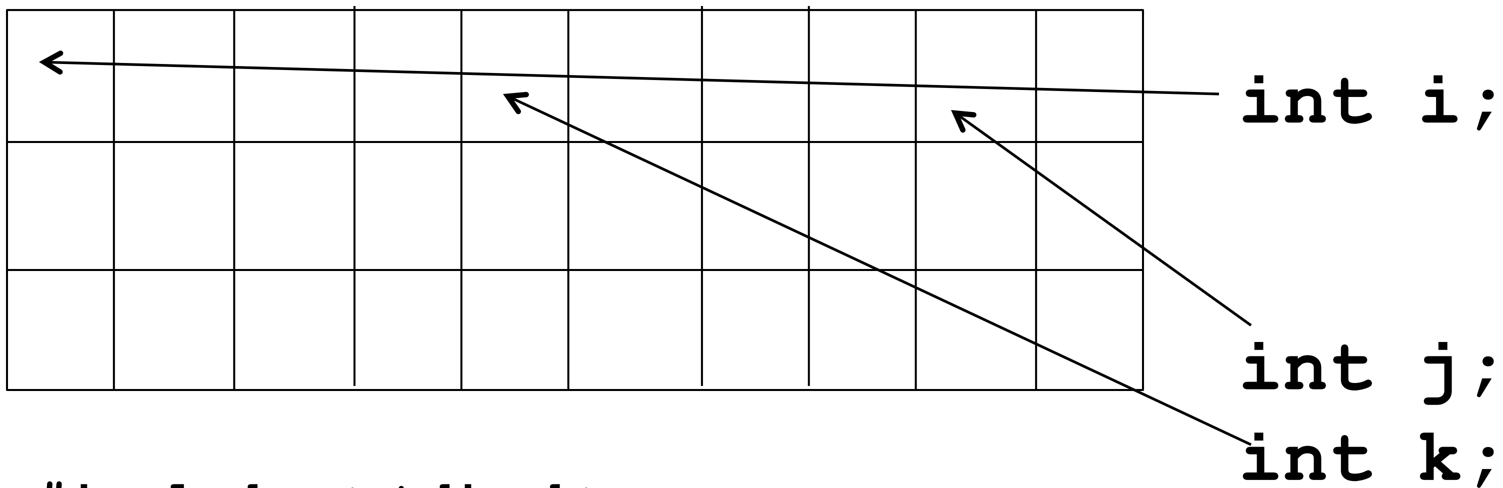
To access the value of the first element of the array:

```
A[0]
*A or *(A+0)
```

To access the value of the sixth element:

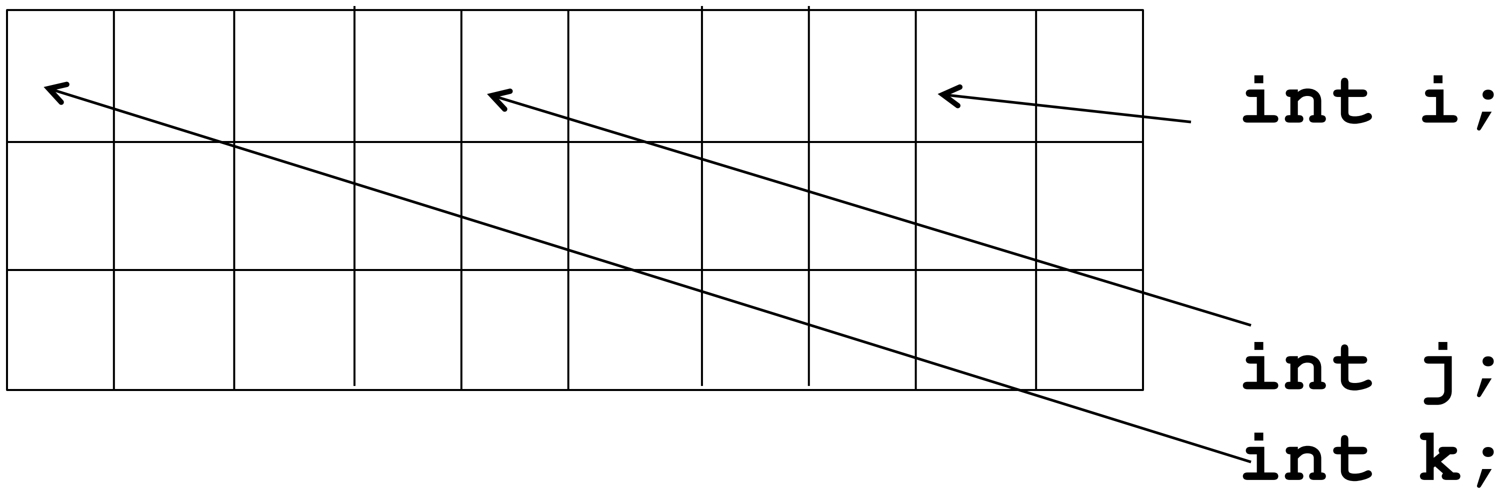
```
A[5]
*(A+5)
```

Random Access Memory




```
#include <stdio.h>
int main(argc, argv)
{
  int i,j,k;
  i = 5; j = 10; k = 15;
  printf("i: value = %d; address = %d\n", i, &i);
  printf("j: value = %d; address = %d\n", j, &j);
  printf("k: value = %d; address = %d\n", k, &k);
}
```

Random Access Memory

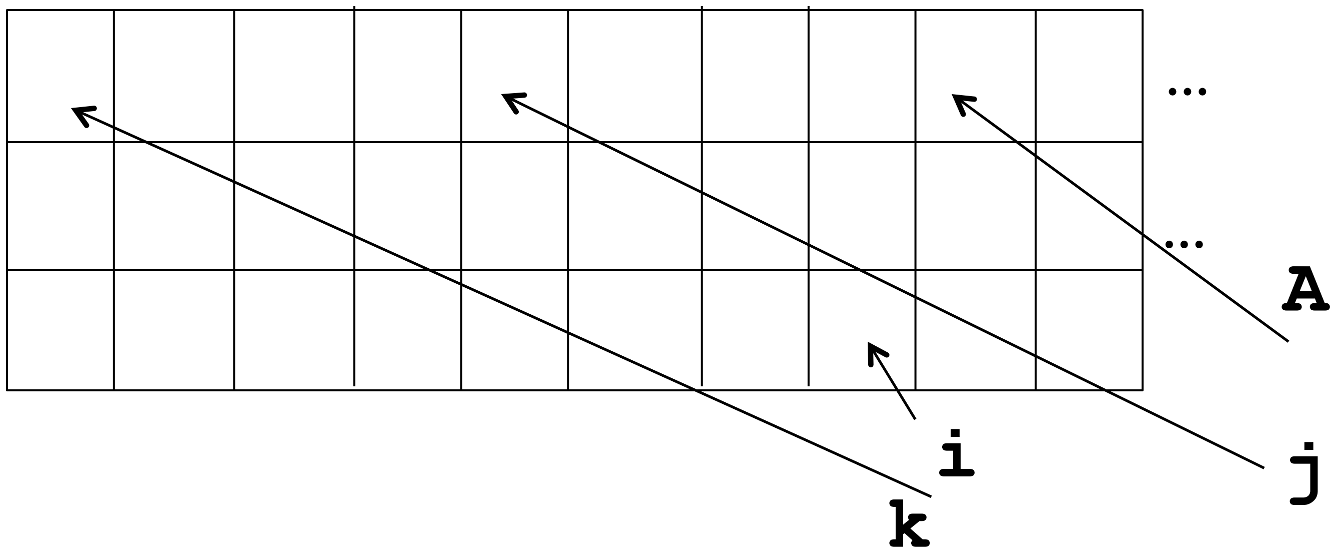



```
i: value =  5; address = 134509652
j: value = 10; address = 134509648
k: value = 15; address = 134509644
```

Random Access Memory


```
int i;
int  A[100];
int j,k;
i = 5; j = 10; k = 15;
printf("i: value = %.2d; address = %d\n", i, &i);
printf("A:                ; address = %d\n", A );
printf("j: value = %.2d; address = %d\n", j, &j);
printf("k: value = %.2d; address = %d\n", k, &k);
```

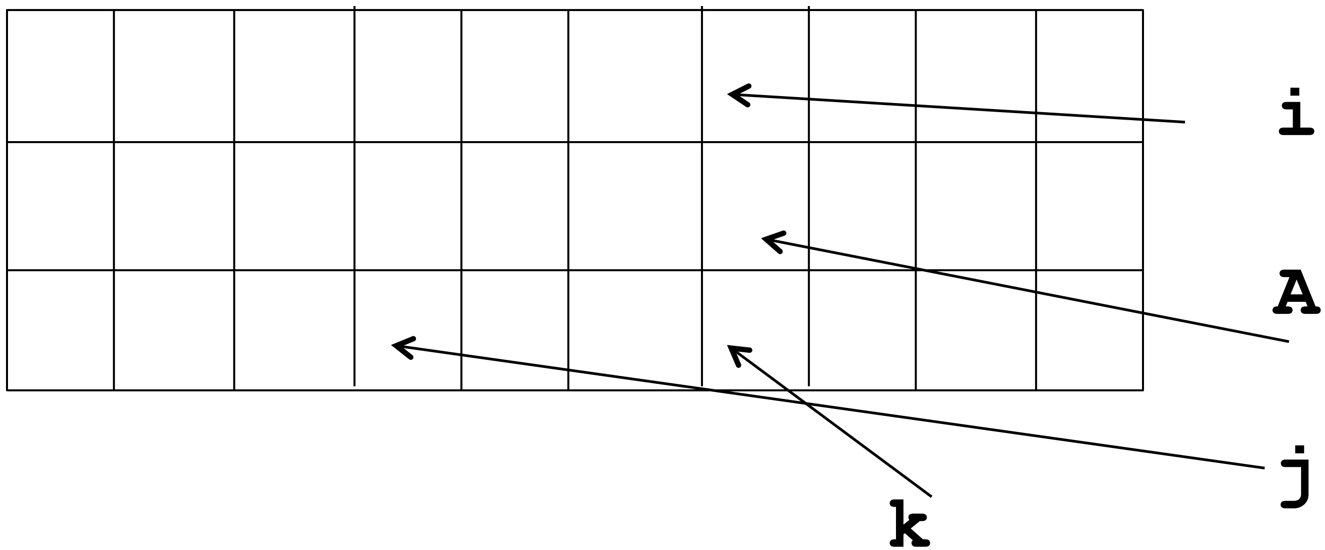
Random Access Memory




```
i: value = 05; address = 134509644
A:           ; address = 134509232
j: value = 10; address = 134509228
k: value = 15; address = 134509224
```

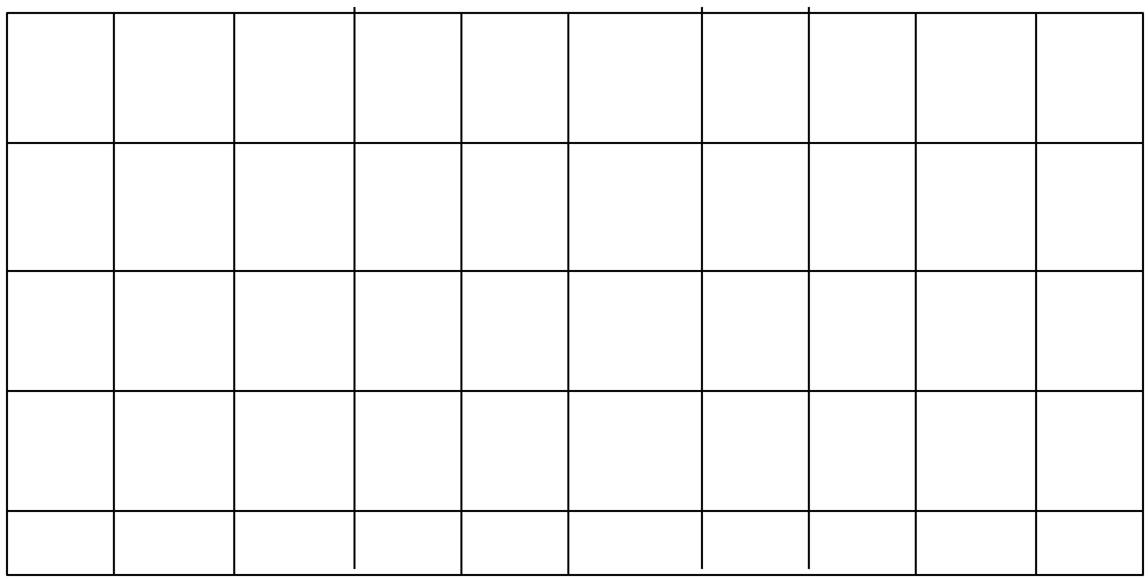
Element address =
Base + index * element_size

Random Access Memory



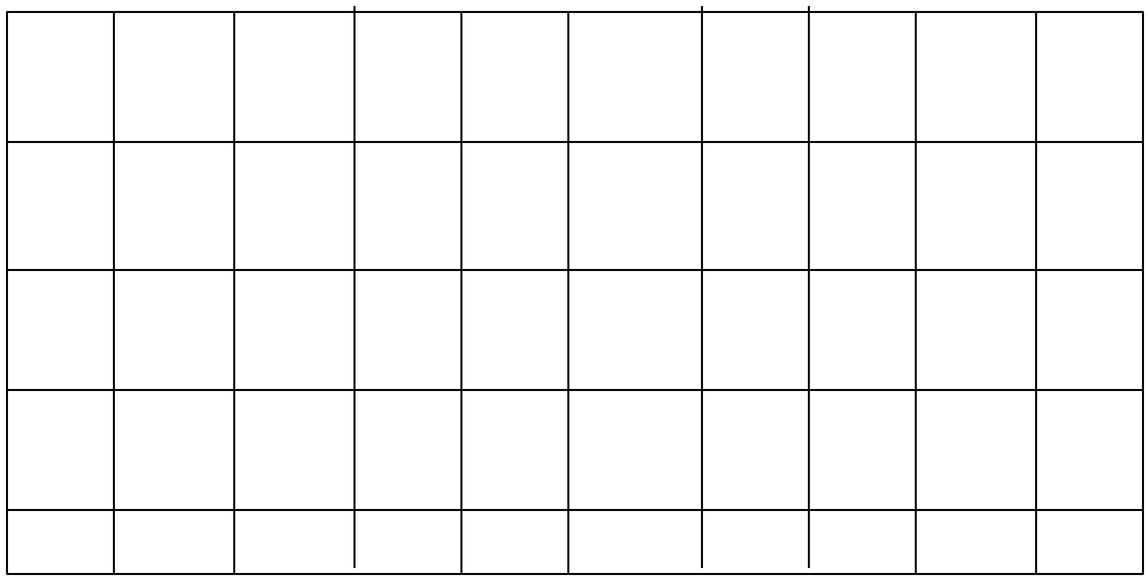
Note: Althought we often draw RAM to *look* like a 2-dimensional array,
it is actually a linear (1-dimensional) array.

A 2-Dimensional Array in C



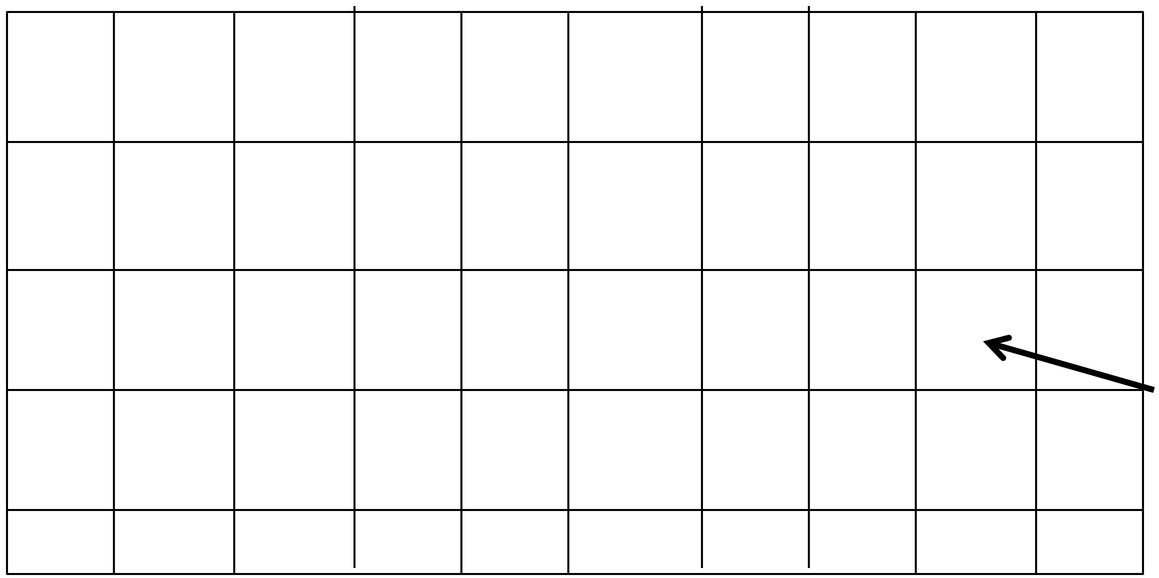
- `int A[rows][cols]`
- `int A[5][10]`

A 2-Dimensional Array in C



- **`int A[5][10]`**
  Actually, **`A`** is an array of size 5,
  each element of **`A`** is an array of 10 ints.

A 2-Dimensional Array in C



`A[2][8]`

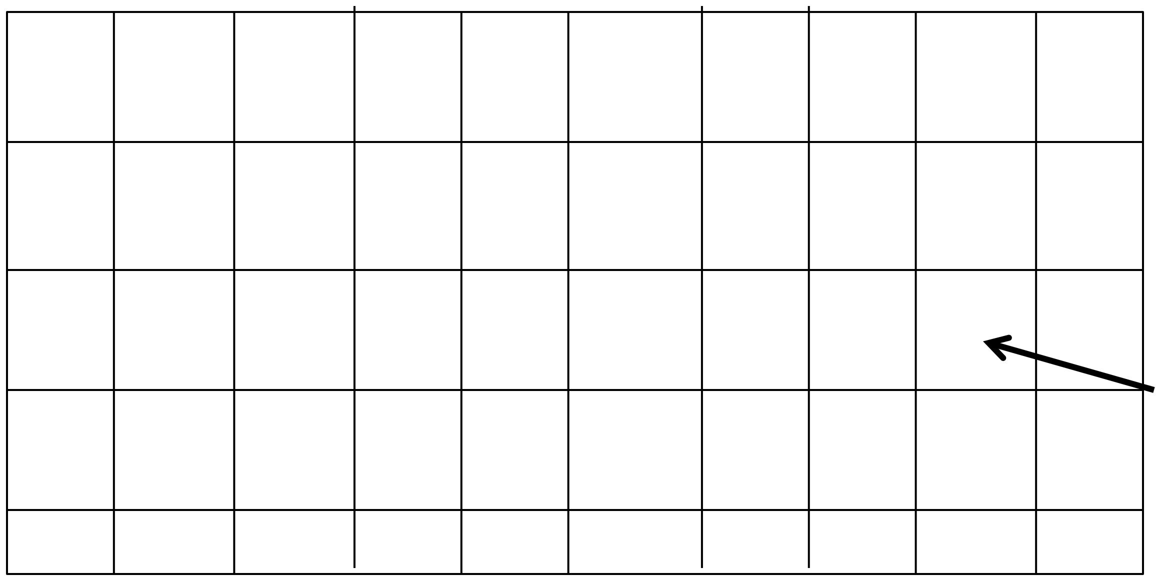
`*(A+2*10*4`
`+8*4)`

- Address of element[thisrow][thiscolumn] =

  Base + thisrow * num_cols * sizeof(element)

+ thiscol * sizeof(element)

A 2-Dimensional Array in C



`A[2][8]`

`*(A+2*10*4 +8*4)`

- Address of element[thisrow][thiscolumn] =

  base + thisrow * num_cols * sizeof(element)

  \+ thiscol * sizeof(element)

- num_cols*sizeof(element) = size of one row

Homework

- What is the difference between:
  - `int a[10][20];`
  - `int *b[10];`
- **?**

- Hint 1: Think memory allocation.
- Hint 2: See K&R section 5.9.
- Hint 3: How could you test your answer?

Back to arrays as dictionaries

- To sort or not to sort?
- How to determine which is best?

Sorting: fine points

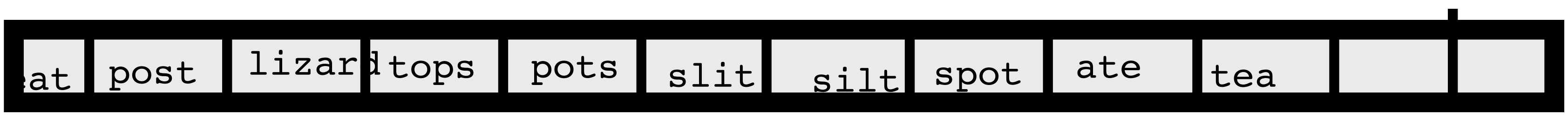
- Sorting assumes the keys are “sortable”, *i.e.* comparable.
  - *e.g.* categories, colors are not sortable, unless you associate an identifier .
- The computer science definition of sorting means to put things into a *well-defined* order.
  - Other ways of sorting: binning, topological sort.

Sorting: fine points

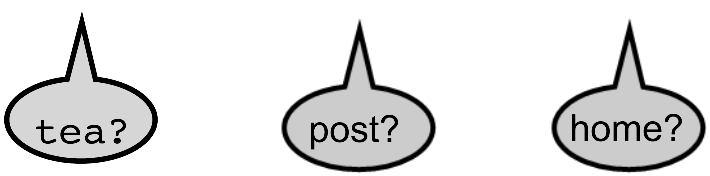
- In computing applications of sorting, there is a *key*, and associated *info*rmation.
- We sort by *key*, and the *info*rmation comes along for the ride.
    - e.g. Student database, sorted on student ID. The information is name, address, degree, *etc*.
- In our examples, we often do not show the *info*rmation explicitly.

Unsorted arrays

- Just put the item in at the end of the array:
  - Insertion is in $O(1)$



- How many comparisons you need to insert?

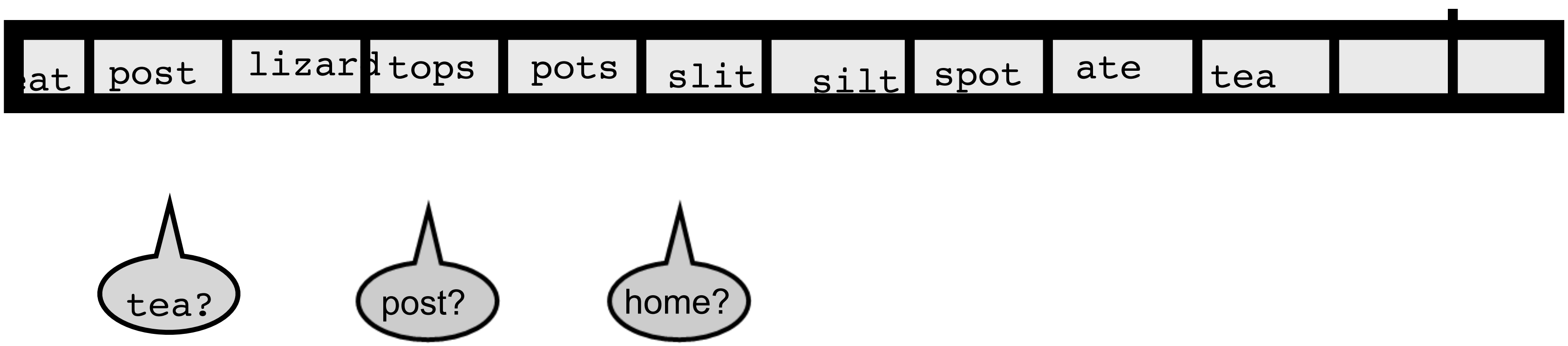


Unsorted array: search

```
for(i=0;A[i]!=EMPTY;i++)   /*A[] is an array of integers */
{
    cmps++;
    if (A[i] == searchkey)
    {
      printf("Found key %d at position %d\n",
             searchkey,i);
      printf("Key comparisons: %d\n", cmps);
      return FOUND;
    }
}
 printf("Key not found\n");
 printf("Key comparisons: %d\n", cmps);
 return NOTFOUND;
```

Unsorted arrays

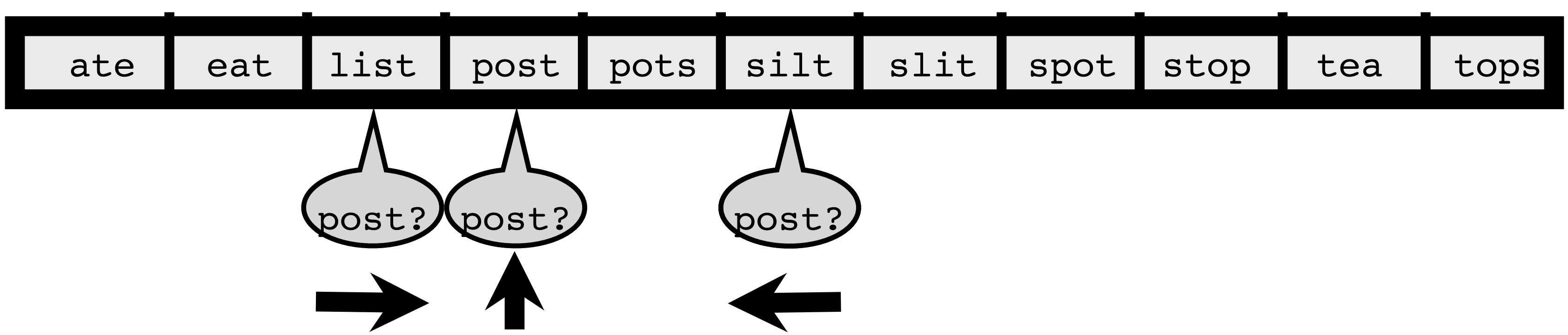
- Just put the item in at the end of the array:
  - Insertion is in $O(1)$



- What is the complexity of search?
  - O(?)

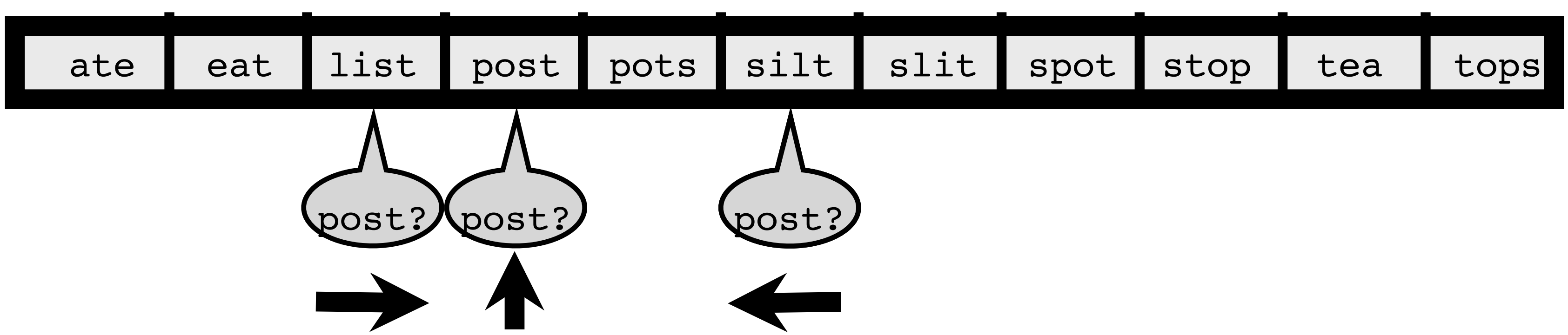
Sorted arrays

- How do you search for “post”?



Sorted arrays

- Binary search in sorted array:



Sorted array: binary search

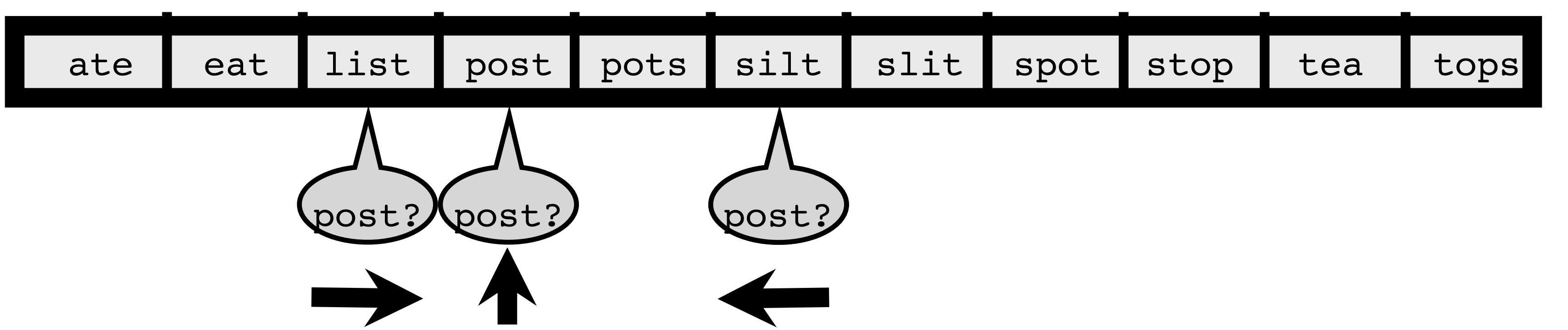
```
int i=0;
int j=ARRAYFILL-1;
while(i<=j)
{
    cmps++; mid = (i+j)/2;
    /*diagnostic*/
    if(DEBUG) printf("i:%d; end:%d; mid:%d\n",i,j,mid);

    if(A[mid]==searchkey)
    {
       printf("Found key %d at position %d, comparisons: %d \n",
               searchkey,i,cmps);
        return FOUND;
    }
    if (searchkey<A[mid] j=mid-1;
    else i=mid+1;
}
printf("Key not found, comparisons: %d\n", cmps);
return NOTFOUND;
```

Searching: analysis

- Unsorted array: O(?)
- Sorted array (binary search): O(?)

Sorted arrays

- Binary search in sorted array:
  - Search is in O(logn)



  - What about insertion?

```
 i=ArrayFill-1;
while(A[i]>INSERTNUM && i>=0)
{
  A[i+1] = A[i];
  i--;
}
A[i+1] = INSERTNUM;  /** only get here if A[i] <=
                      * INSERTNUM, have already moved
                      * previous contents
                      **/
ArrayFill++;  /* to accomodate the new item */

/* test */
for(i=0;i<ArrayFill;i++) printf("%d   ",A[i]);
printf("\n");

/* any assumptions? */
```

Sorted and unsorted arrays

- How to compare?
  - Key comparisons: usually the most expensive operation in searching.

Analysis

- Search:
  - Unsorted array: O(n) per search
  - Sorted array (binary search): O(log n) per search
- Insert:
  - Unsorted array: O(1) per insertion
  - Sorted array: O(n) per insertion

Search: m lookups on n items

- Unsorted array, linear search:
  - n insertions @ 1 operation → n operations O(n)
  - m lookups @ n operations → m*n (worst case)
  - O(n + m*n) = O(mn)
- Sorted array, binary search:
  - n insertions @ n comparisons and n data movements each → $O(n^2)$
  - m searches@ log n comps each -> $m*\log_2 n$
  - $O(n^2 + m \log n)$

Search: m lookups on n items

- Unsorted array, linear search:
  - n + m*n ~ m*n
- Sorted array, binary search:
  - $n^2 + m \log n$

- For m << n, unsorted arrays are better!
- But usually m > n, so use sorted array…
- …or even something better.

Something better?

- What are the worst properties of a sorted array?

Limited size

- We can overcome the limited size problem using dynamic memory allocation.
- C library functions:
  - **`void *calloc(size_t nobj, size_t size)`**
  - **`void *realloc(void *p, size_t size)`**
  - also, of course **`void *malloc(size_t size)`**
  - All defined in **`stdlib.h`**

malloc(): size_t

- **`malloc(size_t size)`**
- **`size_t`** is:
  - an unsigned integer type
  - the type returned by the **`sizeof`** operator
  - widely used in the standard library (**`stdlib`**) to represent sizes and counts.
- *e.g.* **`malloc(sizeof(int))`**

malloc() example (part 1)

```
#define NUMBER 5
int
main (argc, argv)
{
   int   var;
   var = NUMBER;
   printf("%d - %d\n", &var, var);
   return 0;
}

>a.out
134509940 - 5
```

malloc() example: (part 2)

```
#define NUMBER 5
int
main (argc, argv)
{
  int   *ptr;
  ptr =  (int *)malloc(sizeof(int));
  *ptr = NUMBER; /* note '*' */
   printf("%d - %d\n", ptr, *ptr);
  return 0;
}
>a.out
134613280 - 5
```

malloc(): check return value

- Be aware: malloc() can fail!
  - If malloc() fails, it returns NULL.
- Never use a pointer to something where the memory allocation has failed!

```
int *B;
B=(int *)malloc(NUMBER*sizeof(int));
/* always check return value of malloc()*/
If( B == NULL )
{
   printf("malloc() error\n");
   exit(1);
}
```

- Or write a function safemalloc() that does this.

Getting memory for an array using malloc()

```
int A[NUMBER];
/* while insertions < NUMBER array is OK
 * BUT… has a limit
 **/

int *B;
/* always check return value of malloc()*/
if((B=(int *)malloc(NUMBER*sizeof(int)))==NULL)
{
      printf("malloc() error\n");
      exit(1);
}
/** B can now be used like A
 *  better to use calloc(NUMBER,sizeof(int)) */
```

Getting memory for an array using malloc()

```
int A[NUMBER];
/** while insertions < NUMBER array is OK
 * BUT… has a limit
 **/

int *B;
/* always check return value of malloc()*/
if((B=(int *)malloc(NUMBER*sizeof(int)))==NULL)
{
      printf(“malloc() error\n”);
      exit(1);
}
/** B can now be used like A
 * better to use calloc(NUMBER,sizeof(int)) */
```

Getting memory for an array using calloc()

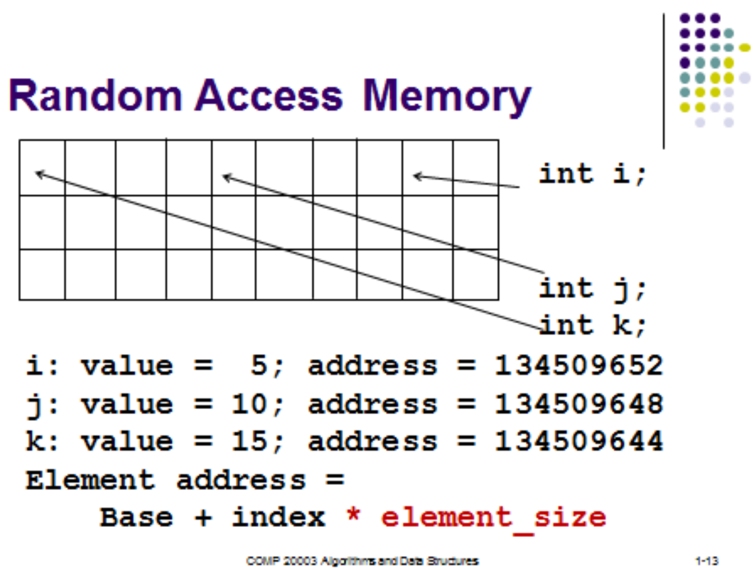
```
int *B;
/* always check return value of malloc()*/
if((B=(int *)calloc(NUMBER,sizeof(int)))==NULL)
{
      printf("malloc() error\n");
      exit(1);
}

/* B now comes with each slot initialized to 0 */
```

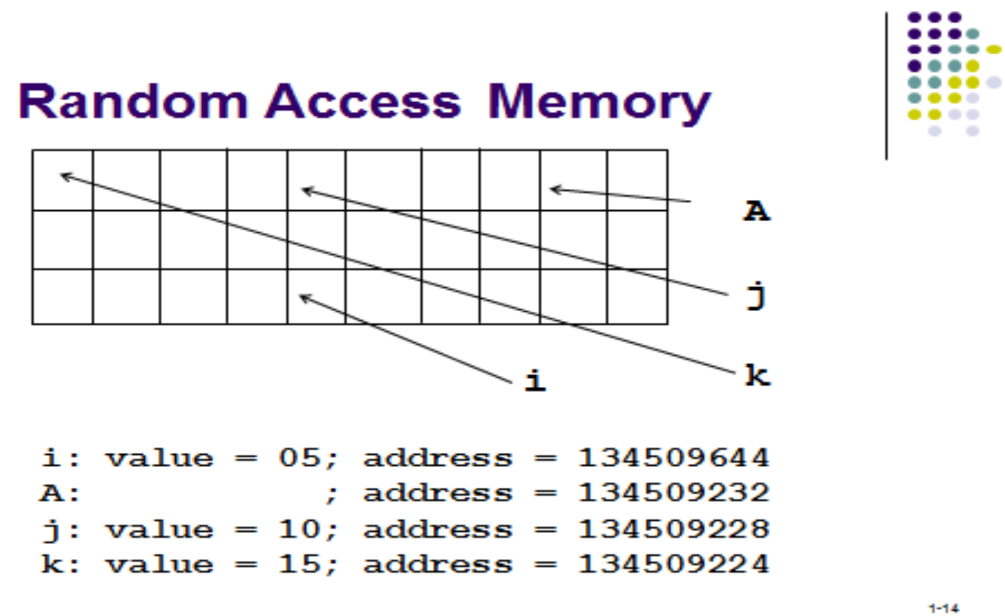
realloc()

```
int *B;
/** as previously, used malloc(),calloc()
 * RESIZE when insertions == NUMBER *
 **/
B = realloc(B,(NUMBER*2)*sizeof(int));
/* should also check realloc()for NULL*/

/* now initialize new part of array */
for(i=NUMBER;i<NUMBER*2;i++)
     B[i] = NULL;
/* now we have a bigger array, first half
copied from the old B  */
```

Details about malloc() and friends

- `malloc()` returns a pointer to a place in memory.
- Argument to `malloc()` specifies how much space to reserve in memory, *i.e.* not allocate to other variables:





What's a pointer?

- A pointer is an address in memory.
- What is the output of this code?

```
int   *ptr;
ptr = (int *)malloc(sizeof(int));
*ptr = 5;
printf("%d, %d", ptr, *ptr);
```

- Now what is the output of *this* code?

```
int   *ptr;
ptr = (int *)malloc(sizeof(int));
ptr = 5;
printf("%d, %d", ptr, *ptr);
```

Details about malloc() and friends

- `#include<stdlib.h>`
- Read the documentation for fine points:
  - `malloc()` returns uninitialized space
  - `calloc()` returns space initialized to 0
  - `realloc(void *p, size_t size)` returns space where the start is copied from p and the rest is unintialized.
- Check return value of all memory alloc functions.

malloc() and free()

- **`malloc()`** allocates memory.
- **`free()`** use to deallocate memory

```
void   *ptr;
ptr = malloc(NUMBER_OF_BYTES);
/* do things  until finished with the
   contents pointed to by ptr */
free(ptr);
```

Back to sorted arrays…

- Space limitations:
  - Can use `realloc()`.
  - Or can use linked list (sorted linked list).

Pointers

- For an excellent exposition of pointers in C, see the excellent tutorial by Ted Jensen:
  - LMS Resources → Pointers and Arrays in C

Pointers

- A pointer in C is an address.

```
int k;
int *ptr;

k=5;
ptr = &k;
printf("%d", *ptr);

<5>
```

Pointers

- A pointer in C is an address.
- * is the dereferencing operator.

```
int k;
int *ptr;

k=5;
ptr = &k;
printf("%d", *ptr);

<5>
```

Pointers

- A pointer in C is an address.
- Where A is the name of an array, A is a pointer to the array, and
  - A[0] is equivalent to *(A+0),
  - A[5] is equivalent to *(A+5)…

Memory Allocation: Summary

- **`malloc()`,`calloc()`,and `realloc()`** return:
  - The (untyped) address of allocated memory;
  - *i.e.* a pointer to allocated memory.

```
struct node *ptr; /*allocates just enough
         room for an address */
ptr = (struct node *)malloc(sizeof(struct
node)); /* allocates enough room for the
         node */
```

- We have discussed the strong and weak points of sorted arrays as search structures:

Linked lists: flexibility, but overheads

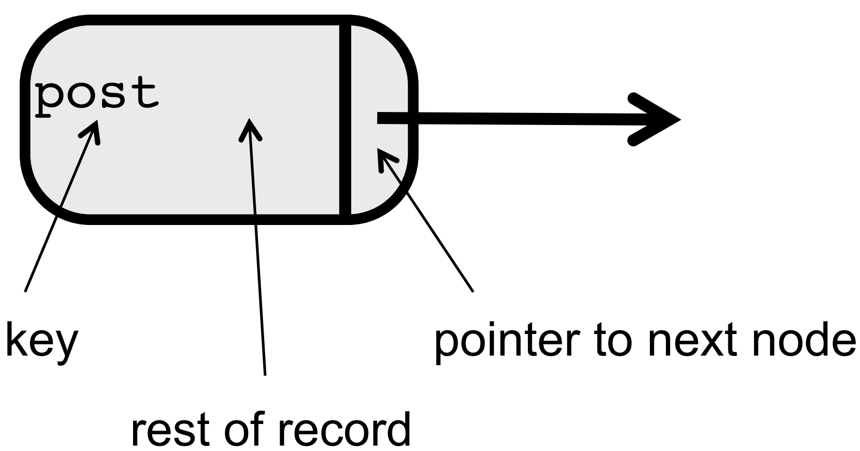
- In a linked list, each item (or key) is located in an arbitrary place in memory, with a link (pointer) to the next item.
- If arbitrary order, finding item is still $\Theta(n)$-time.
- Once insertion point has been determined, easy to insert (or delete) a new item, by rearranging links.

Linked lists: flexibility, but overheads

- Takes extra space for each item in the list.
- Takes extra time to allocate the memory for the node for each item.

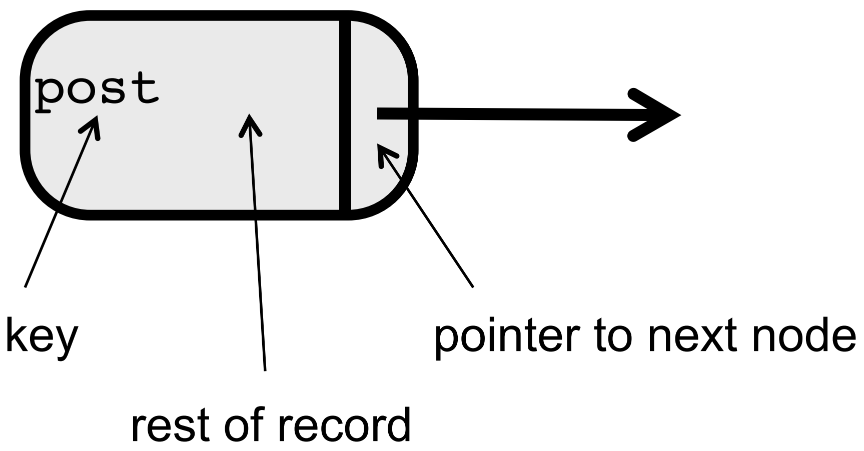
The node

```
struct node{
        record r;
        struct node *next;};
```



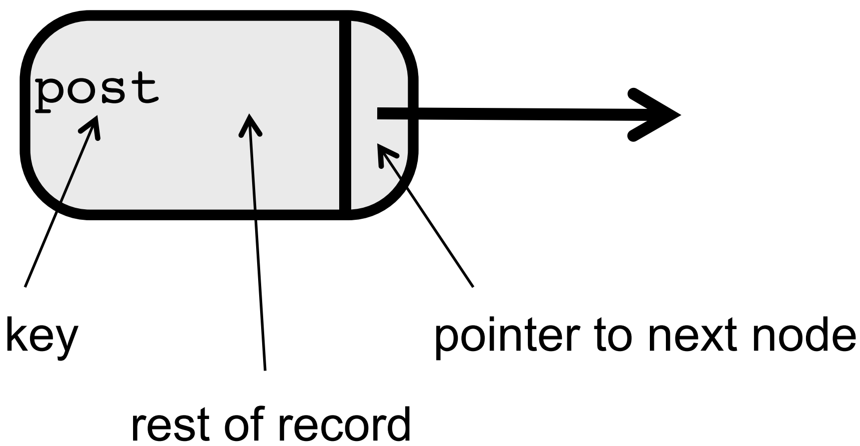
The node

```
typedef
struct node{
        record r;
        struct node *next;}
node_t;
```

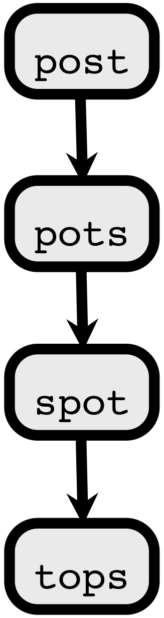


The node

```
typedef
struct node{
     record r;
     struct node *next;}
node_t;
```

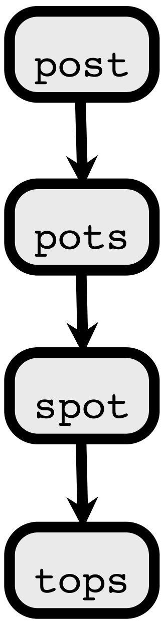


A linked list of nodes



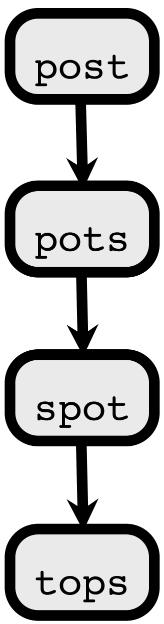

```
struct node
{
    char  *key;
    char  *info;
    struct node  *next;
};

struct node   *newnode;

newnode = /*malloc space and
put in the key and info */
/* suggested declaration style for
beginner and intermediate */
```

A (sorted) linked list of nodes




```
OR
typedef
struct node{
     record r;
     struct node *next;}
node_t;

typedef node_t  *node_ptr;

node_ptr   newnode;

/* more advanced style */
```

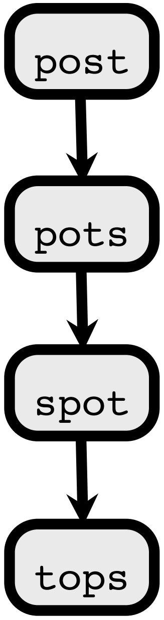
A (sorted) linked list of nodes




```
struct node
{
    char  *key;
    struct node  *next;
};

struct node   *newnode;

newnode = /*malloc space
and put in the key and
info */
```

```
/* suggested declaration style for
   beginner and intermediate */
```

Traverse the list



Traverse the list

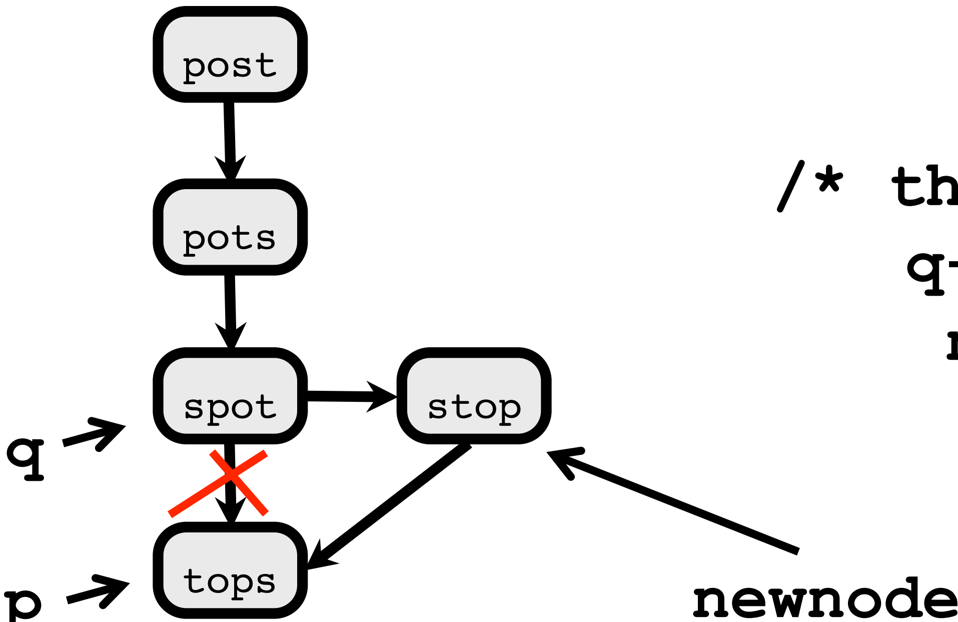
```
p = listhead;
if(p==NULL) /* empty list */
else
  while( p->next !=NULL)
  {
        printf("%d\n", p->key);
        p = p->next;
  }
  printf("%d\n",p->key);
```

Traverse the list

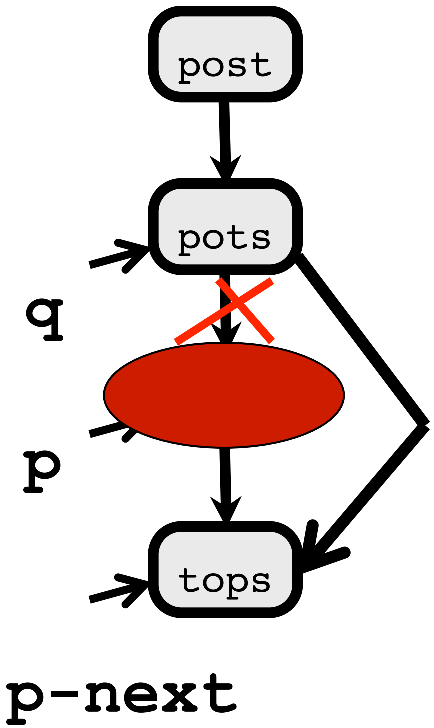
```
p = listhead;
if(p==NULL)
{
   printf("List empty\n");
   return;
}

/* traverse and print key */
 while(p->next !=NULL)
 {
    printf("%d\n", p->key);
    p = p->next;
 }
 printf("%d\n",p->key);  /* why? */
 return;
```

Inserting a new node into a sorted linked list

```
/* find correct place in list
          to insert */
       /* how? */
/* then adjust pointers*/
  q->next = newnode;
  newnode->next=p;
```



Deleting a node




```
/* p points to the node
we want to delete */
q->next = p->next;
free(p);
```

Linked lists: sorted *vs.* unsorted

- What are the advantages of keeping a linked list in sorted order?

- What are the disadvantages?

Search: Arrays vs. Linked Lists

- Sorted arrays:
  - Fast search (binary search), but
  - Slow insertion (keeping sorted order).
- Sorted array:
  - Fixed size, but
  - Can grow with realloc()
  - Usual growth is factor of 2
- Array needs only one memory allocation.
- Linked list needs many.

Table of "running times"

	One Search	One Insert
Unsorted array		
Sorted array		
Unsorted linked list		
Sorted linked list		

Table of "running times"

	One Search	One Insert
Unsorted array	n	1
Sorted array	$\log n$	n
Unsorted linked list	n	1
Sorted linked list	n	n

Exercize

How many operations are needed for *m* searches in a dictionary of *n* items?

	Each Insertion	Each Search	Build + Search
Unsorted array	O(1)	O(*n*)	***O(mn)***
Sorted array	O(*n*) comps + O(*n*) data movements	O(log *n*)	
Unsorted linked list	O(1)	O(*n*)	
Sorted linked list	O(n) comps	O(*n*)	

Practical complexity and algorithms

- O(1): Execute instructions once (or a few times), independent of input.
  - Example: pick a lottery winner.
- O(log n): keep splitting the input, and only operate on *one* section of the input.
  - Example:
- O(n): Execute instruction(s) once for each data item:
  - Example:

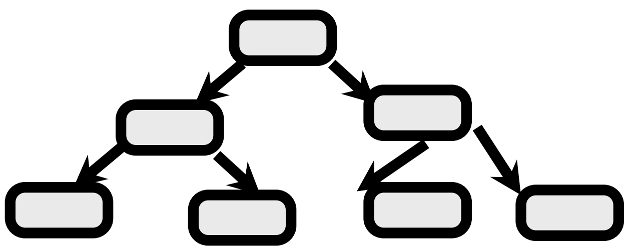
Practical complexity and algorithms

- $O(n \log n)$: split the input repeatedly, and do something to *all* the segments
  - Example: Many sorting algorithms
- $O(n^2)$: For each item, do something to all the others. (Nested loops.)
  - Example:
  - Note: getting slow for large data…
- $O(n^3)$:
- $O(2^n)$:

Breaking out of linearity

- Compare:
  - Linked list

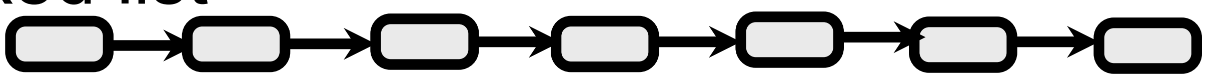
  - Binary tree



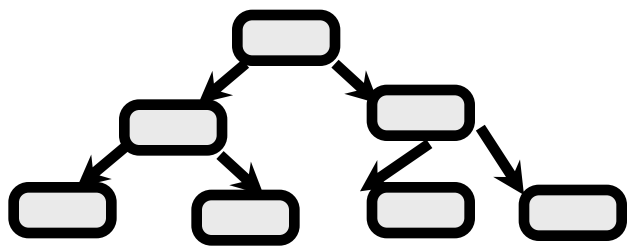
- If we reliably know whether the desired item is in the left subtree or the right subtree, we could find it more quickly.

Breaking out of linearity

- Compare:
  - Linked list



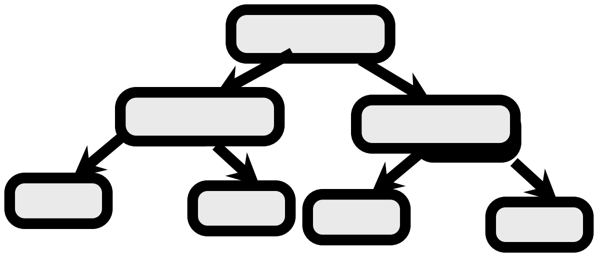
  - Binary tree



- Note for a complete binary tree, half the nodes are at the bottom level…

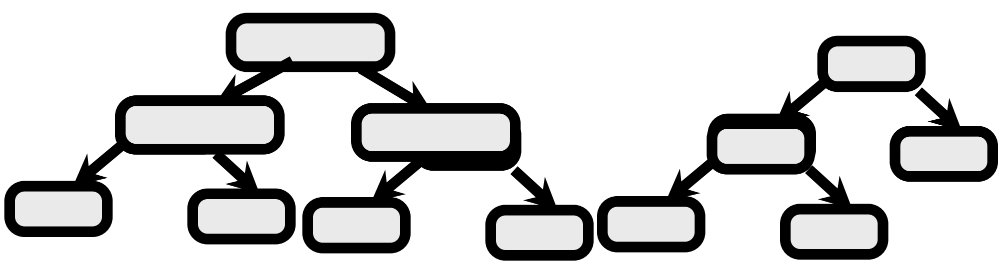
What is a complete binary tree?

- A **complete** binary tree is a binary tree in which **every level**, **except** possibly the **last**, is completely filled, and **all nodes are as far left** as possible.



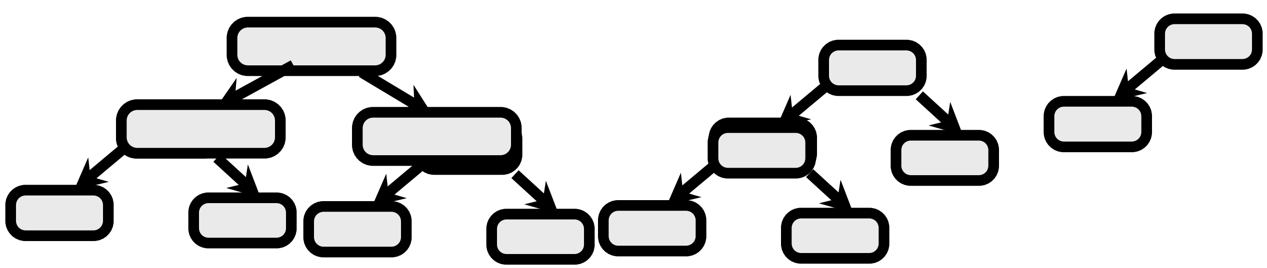
What is a complete binary tree?

- A **complete** binary tree is a binary tree in which **every level**, **except** possibly the **last**, is completely filled, and **all nodes are as far left** as possible.



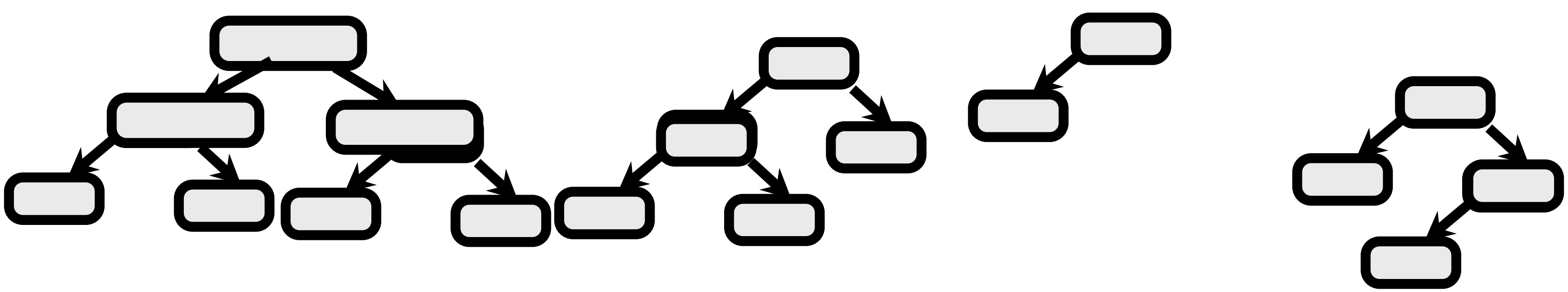
What is a complete binary tree?

- A **complete** binary tree is a binary tree in which **every level**, **except** possibly the **last**, is completely filled, and **all nodes are as far left** as possible.

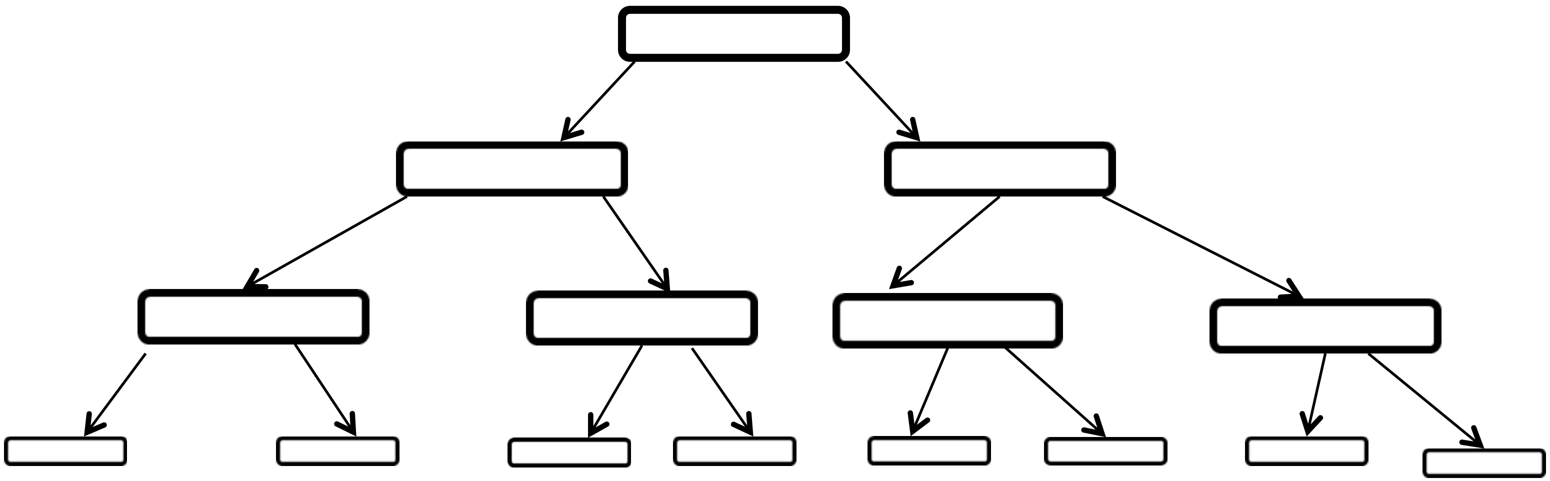


What is a complete binary tree?

- A **complete** binary tree is a binary tree in which **every level**, **except** possibly the **last**, is completely filled, and **all nodes are as far left** as possible.



Looking at a complete binary tree



Nodes

2^0

2^1

2^2

2^3

....

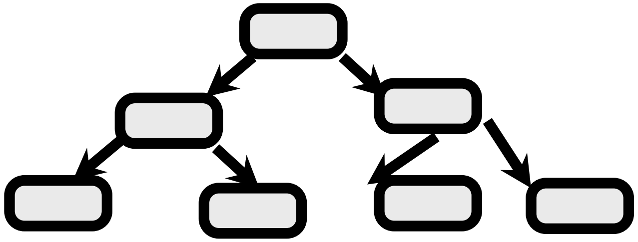
$\sim 2^{\log n}$

A complete binary tree of n nodes has depth approximately $\log_2 n$.

Breaking out of linearity

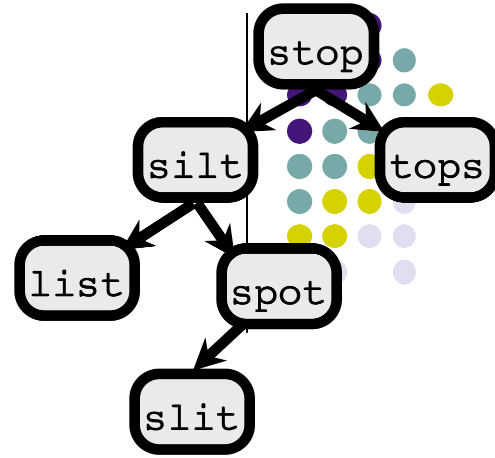
- Compare:
  - Linked list

  - Binary tree



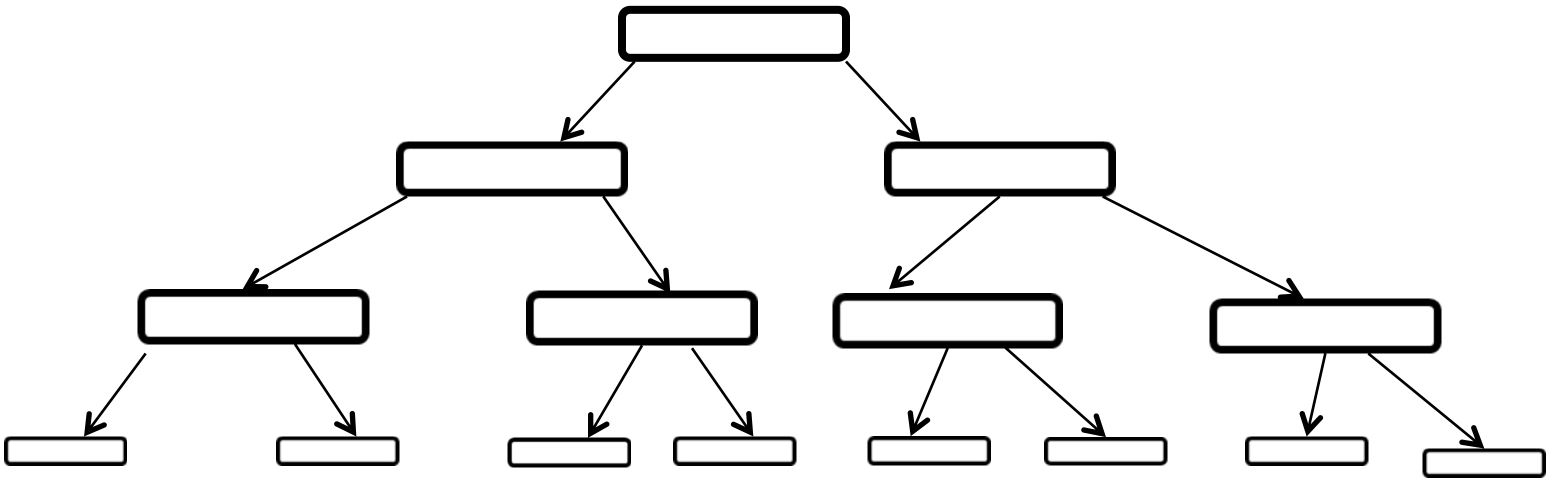
- As we will see, both insertion and search are log *n*-time operations.

How a binary search tree work?



- In a sorted linked list, **`next`** links to a record with a key ≥ this one.
- In a BST, **`left`** links to items with key < current key
- **`right`** links to items with key ≥ current key.
- Find node with key **`slit`** in this tree.

Looking at a complete binary tree



Nodes

2^0

2^1

2^2

2^3

….

$2^{\log n - 1}$

- For a complete binary tree of n nodes, to get to the bottom will take not more than $\log_2 n$ key comparisons.
- Contrast with linked list, to get to the end….

Binary tree exercizes

- Put the following numeric keys into a bst:
  - 45, 37, 86, 90, 50, 16, 37
  - How long (how many key comparisons) does it take to search for key=5?
- Put the following numeric keys into a bst:
  - 90, 86, 50, 45, 37, 32, 16
  - How long does it take to search for key=5?

Best case run time in bst: Perfectly balanced tree

- Best case for bst: perfectly balanced.
- Height of tree with n items:$\log_2 n$.
- Path from root to any node:
  - Maximum length: $\log_2 n$
  - Average length: also $\log_2 n$
- Insertion/search/deletion are all O(logn) for a well-balanced tree.

Average case run time in bst

- Average case for insertion/search/deletion in a binary search tree is also O(logn).

- More exactly ~1.4 log n.

Worst case run time in bst: Stick

- Worst case for bst: a stick.
  - e.g. when items are inserted in sorted order.
- The bst degenerates to a linked list!
- Height of tree with n items: n
- Path from root to any node:
  - maximum length: n
  - average length: n/2
- Insertion/search/deletion are O(n)!

Binary search trees

- Deletion?

Something to think about

- Why don’t we just randomize the order of the items we insert into the binary search tree, to prevent worst case behavior?

Binary search trees

- Good average case behavior – logn.
- Bad worst case behavior – n.
- So overall bst O(n).
  - Actual behavior usually not linear.
  - But potentional linearity.
- Balanced trees: AVL, red-black; 2,3,4; B +tree.

Dictionaries: Summary

- We have looked at various underlying data structures for implementing dictionaries:
  -
  -
  -
  -
  -

Dictionaries: Summary

- We have analyzed the computational complexity for these data structures:
  -
  -
  -
  -
  -

Dictionaries: Summary

- So far the best we have done is log n search, where either:
  - Insertion is O(n); or
  - O(log n) average case but O(n) worst case.
- We can do better…

Next section

- Balanced trees.