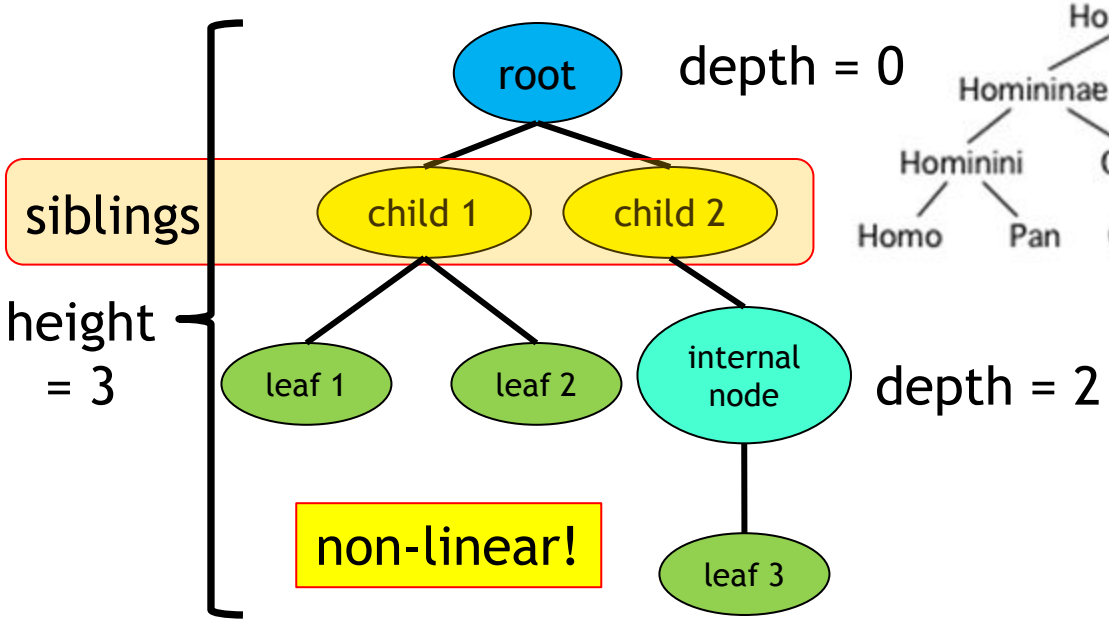
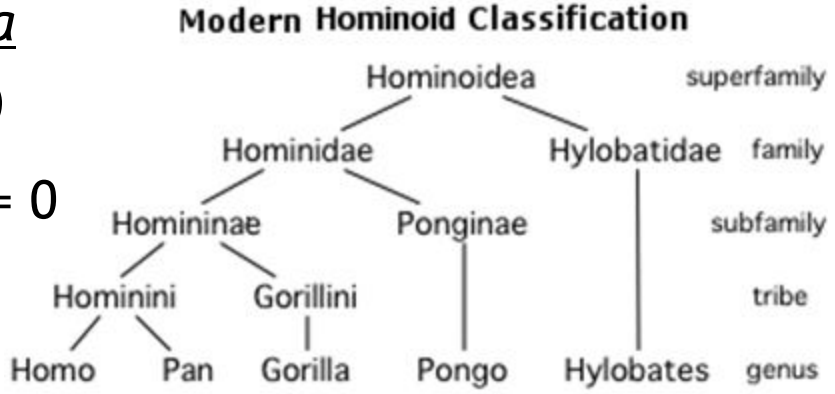
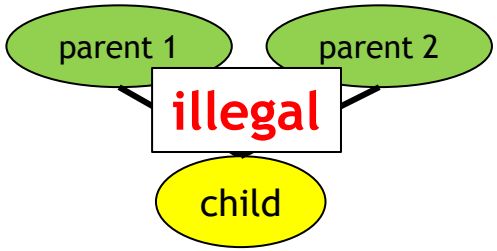
Trees

- A data structure for *hierarchical data*
  - data stored in nodes (or vertices)





- Every node except the root has exactly one parent



Vocabulary

1. Root node
2. Leaf node
3. Edge
4. Child/parent
5. Siblings
6. Internal node
7. Ancestor/descendent
8. Depth of a node
9. Height of a tree

Binary Trees

- A binary tree is a tree where each internal node has at most two children
  - sometimes indicated as left-child and right-child

- Implementation:

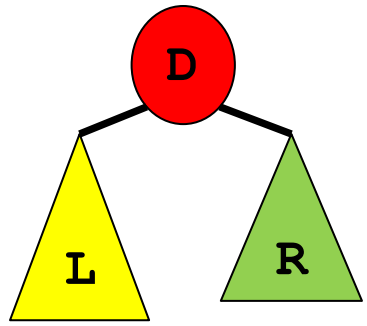
```
typedef struct node {
  int data; /* or some other type */
  struct node *left;
  struct node *right;
} tNode;
```

- If an internal node only has one child, **`left`** or **`right`** will be **`NULL`**
  - for a leaf, **`left = right = NULL`**

- To access: **`tNode *root; /* NULL, if tree is empty */`**

Creating a node

- Suppose we want to create a new node containing some data D
  - the left child should be the sub-tree L
  - the right child should be the sub-tree R
  - L and R could be empty

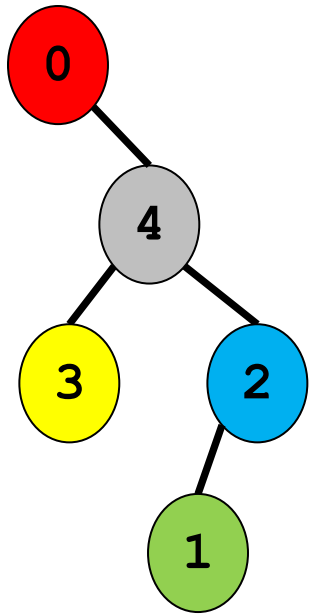



```
tNode *makeNode(int D, tNode *L, tNode *R) {
  tNode *node = (tNode *) malloc(sizeof(tNode));
  if(node == NULL) { return NULL; }
  node->data = D;
  node->left = L;
  node->right = R;
  return node;
}
```

Question

- What kind of a tree does the following code make?

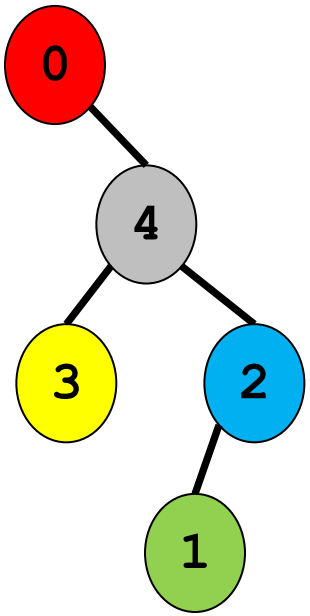
```
tNode *makeNode(int D, tNode *L, tNode *R) {
  /* code from previous slide */
}

void main() {
  tNode *n1, *n2, *n3, *n4, *root;
  n1 = makeNode(1, NULL, NULL);
  n2 = makeNode(2, n1, NULL);
  n3 = makeNode(3, NULL, NULL);
  n4 = makeNode(4, n3, n2);
  root = makeNode(0, NULL, n4);
}
```



Simplified forall function for trees

- Suppose we want to print all nodes in a tree:

```
void forall(tNode *node) {
  if(node == NULL) { return; }
  printf("%d\n", node->data);
  forall(node->left);
  forall(node->right);
}
```

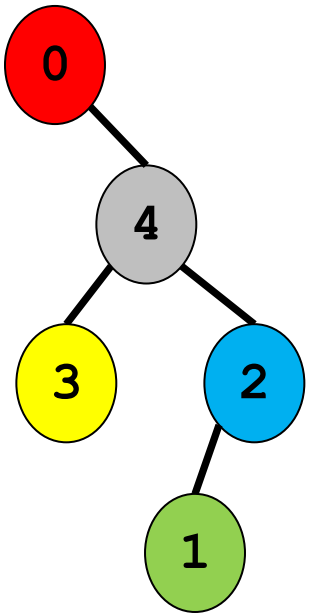


- Question 1: What happens on this?
- Question 2: How about with this change?
  - first version: preorder traversal (this, left, right)
  - second version: inorder traversal (left, this, right)
  - also: postorder traversal (left, right, this)

Simplified forall function for trees

- Suppose we want to print all nodes in a tree:

```
void forall(tNode *node) {
  if(node == NULL) { return; }
  printf("%d\n", node->data);
  forall(node->left);
  forall(node->right);
}
```



- Question 1: What happens on this?
- Question 2: How about with this change?
  - first version: preorder traversal (this, left, right)
  - second version: inorder traversal (left, this, right)
  - also: postorder traversal (left, right, this)

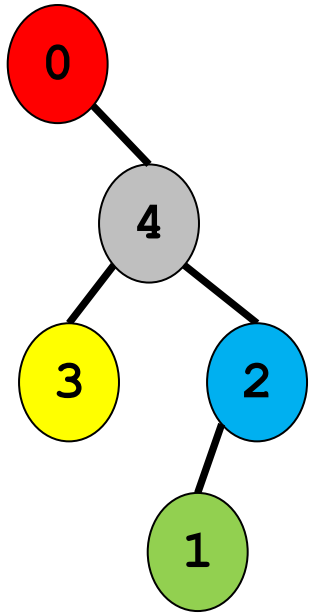
General preorder

```
tNode *preorder(tNode *node,
                int(*task)(int, void*), void*),
                void *args) {
  if(node == NULL) { return NULL; }
  if(!task(node->data, args)) { return node; }
  tNode *temp = preorder(node->left, task, args);
  if(temp != NULL) { return temp; }
  return preorder(node->right, task, args);
}
```

Question

- What does this function compute?

```
int strange(tNode *node) {
  if(node == NULL) { return 0; }
  return strange(node->left)
       + node->data
       + strange(node->right);
}
```



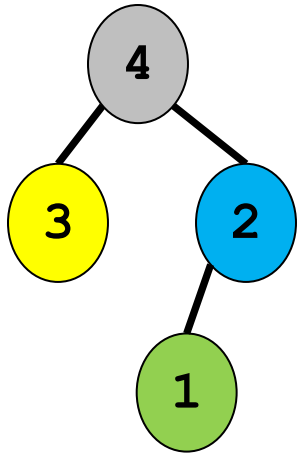
- *For the example*: 0 + 3 + 4 + 1 + 2
  - in general, sum of data

- **Question**: The above function uses inorder traversal. What would happen if we used preorder traversal? Postorder?

Freeing memory

- The following function frees memory used by a tree:

```
void freeTree(tNode *node) {
   if(node == NULL) { return; }
   freeTree(node->left);
   free(node);
   freeTree(node->right);
}
```

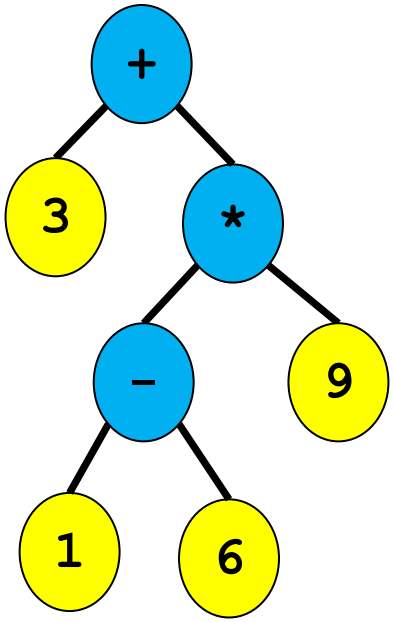


- **Question:** The above function uses postorder traversal. What would happen if we used inorder traversal? Preorder?

An application: expression trees

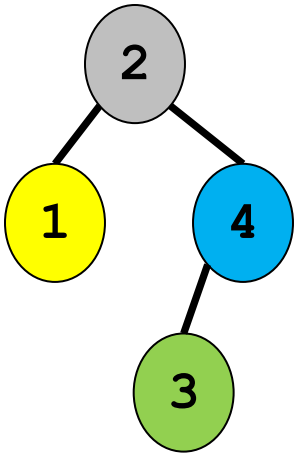
- Evaluating the tree:

```
// assume node != NULL
int eval(tNode *node) {
  if(isInt(node->data)) {
    return toInt(node->data);
  }
  if(isPlus(node->data)) {
    return eval(tNode->left)
         + eval(tNode->right);
  }
  if(isMult(node->data)) { ... }
  if(isUMinus(node->data)) {
    return -eval(node->left);
  }
  // never reach here
}
```



Ordered Binary Trees

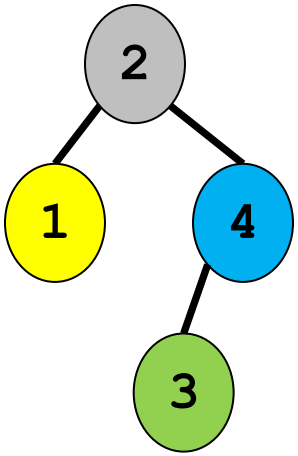
- Suppose the data-type used in tree nodes supports ordering
  - e.g., any two **`int`**s can be compared using **<**
- We can then define an ordered binary tree where ***every*** node *x* satisfies this *ordering property*:
  - the data at all nodes in *x*'s left sub-tree is less than or equal to the data at node *x*
  - the data at all nodes in *x*'s right sub-tree is greater than the data at node *x*
- Finding data in a tree
  - without the ordering property
  - with the ordering property



Finding data in an ordered tree

- This function assumes that **node** is the root of an ordered tree

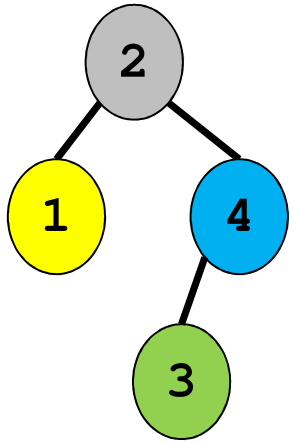
```
int find(tNode *node, int D) {
  if(node == NULL) { return 0; }
  if(D == node->data) {
    return 1;
  } else if(D < node->data) {
    return find(node->left, D);
  } else {
    return find(node->right, D);
  }
}
```



Finding data in an ordered tree

- This function assumes that **node** is the root of an ordered tree

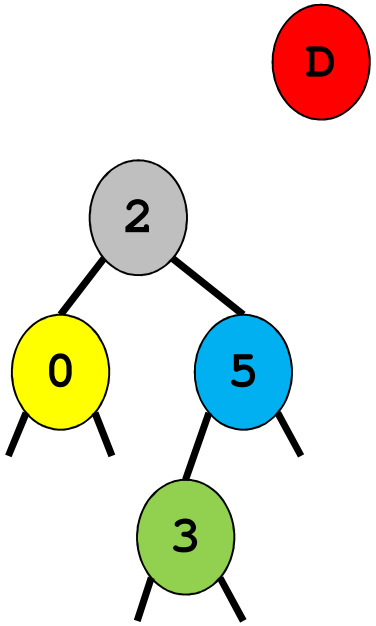
```
int find(tNode *node, int D) {
  if(node == NULL) { return 0; }
  if(D == node->data) {
    return 1;
  } else if(D < node->data) {
    return find(node->left, D);
  } else {
    return find(node->right, D);
  }
}
```



Inserting data into an ordered tree

- Our ordered tree data structure will not permit duplicate data
  - can be modified to allow duplicates

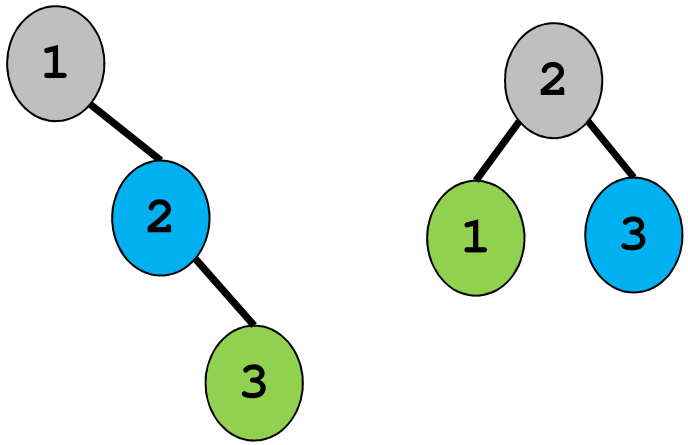
```
tNode *insert(tNode *node, int D) {
  if(node == NULL) {
    return makeNode(D, NULL, NULL);
  }
  if(D < node->data) {
    node->left =
      insert(node->left, D);
  } else if(D > node->data) {
    node->right =
      insert(node->right, D);
  }
  return node;
}
```



Understanding insert

- Draw the tree that will be created

```
tNode *insert(tNode *node, int D) {
  /* code from previous slide */
}

void main() {
  tNode* root =
    insert(
      insert(
        insert(NULL, 2),
           3),
        1);
}
```



Question 1: Worst-case running time?
Question 2: What if **`makeNode`** fails?

A better insert

```
tNode *insert(tNode *node, int D) {
  if(find(node, D)) { return NULL; }
  tNode *newNode = makeNode(D, NULL, NULL);
  if(newNode == NULL) { return NULL; }
  return rec_ins(node, newNode);
}

tNode *rec_ins(tNode *node, tNode *newNode) {
  if(node == NULL) { return newNode; }
  if(newNode->data < node->data) {
    node->left = rec_ins(node->left, newNode);
  } else if(newNode->data > node->data) {
    node->right = rec_ins(node->right, newNode);
  }
  return node;
}
```

If **newNode** is the root of an ordered tree, will **rec_ins** merge the two ordered trees?

Merging two trees

- Write a function to merge all nodes from **`src`** into **`dest`**

```
tNode *merge(tNode *src, tNode *dest) {
  if(src == NULL) { return dest; }
  dest = insert(dest, src->data);
  dest = merge(src->left, dest);
  return merge(src->right, dest);
}
```

- The above code traverses **`src`** nodes in preorder
  - what if we used inorder? Postorder?

- If both trees have *n* nodes initially, how long will **`merge`** take?

Deleting data from an ordered tree

- Recall that our ordered trees do not permit duplicates

```
tNode *delete(tNode *node, int D) {
  if(node == NULL) { return NULL; }
  if(D < node->data) {
    node->left = delete(node->left, D);
  } else if(D > node->data) {
    node->right = delete(node->right, D);
  } else {
    tNode *oldNode = node;
    node = merge(node->left, node->right);
    freeTree(oldNode->left);
    free(oldNode);
  }
  return node;
}
```

Slow because it:

- does not recycle old nodes
- wastefully compares data in **`node->left`**, **`node->right`** subtrees

What does this code do?

```
tNode *extractMax(tNode *node, tNode *ans) {
  // assume node != NULL
  if(node->right == NULL) {
    ans->data = node->data;
    tNode *temp = node->left;
    free(node);
    return temp;
  }
  node->right = extractMax(node->right, ans);
  return node;
}

tNode *extractMin(tNode *node, tNode *ans) {
  /* similar */
}
```

A better delete

```
tNode *delete(tNode *node, int D) {
  if(node == NULL) { return NULL; }
  if(D < node->data) {
    node->left = delete(node->left, D);
  } else if(D > node->data) {
    node->right = delete(node->right, D);
  } else {
    if(node->left != NULL)
      node->left = extractMax(node->left, node);
    else if(node->right != NULL)
      node->right = extractMin(node->right, node);
    else { free(node); return NULL; }
  }
  return node;
}
```

Exam 1: Review

- **Question 1 (4 points)**: What is the error?

```
1: void freeList(List list) {
2:   Node *this = (Node *) list;
3:   if(this != NULL)
4:     freeList(this->next);
5:   free(this);
6: }
```

Bad typecast?
No base-case?
Type mismatch?
Segmentation fault?

- **Question 2 (6 points)**: Worst-case time complexity for size *n* list:

```
ListEntry previous(List list, ListEntry e);
```

ArrayList1	$O(1)$
LinkedList 1	$O(\log n)$
	$O(n)$

Question 3 (10 points)

```
int sizeOf(DLL_node *node) {
  // assume node != NULL;
  /* Idea: start from node and go
     backwards until NULL, and go
     forwards until NULL, counting
     as you go
  */
  int ans = 1;
  for(DLL_node *p=node->prev; p != NULL; p = p->prev)
    ans++;
  for(DLL_node *p=node->next; p != NULL; p = p->next)
    ans++;
  return ans;
}
```

Question 4 (5+5 points)

```
int peek(Queue);  // returns data at front of queue

first = peek(myQueue);
```

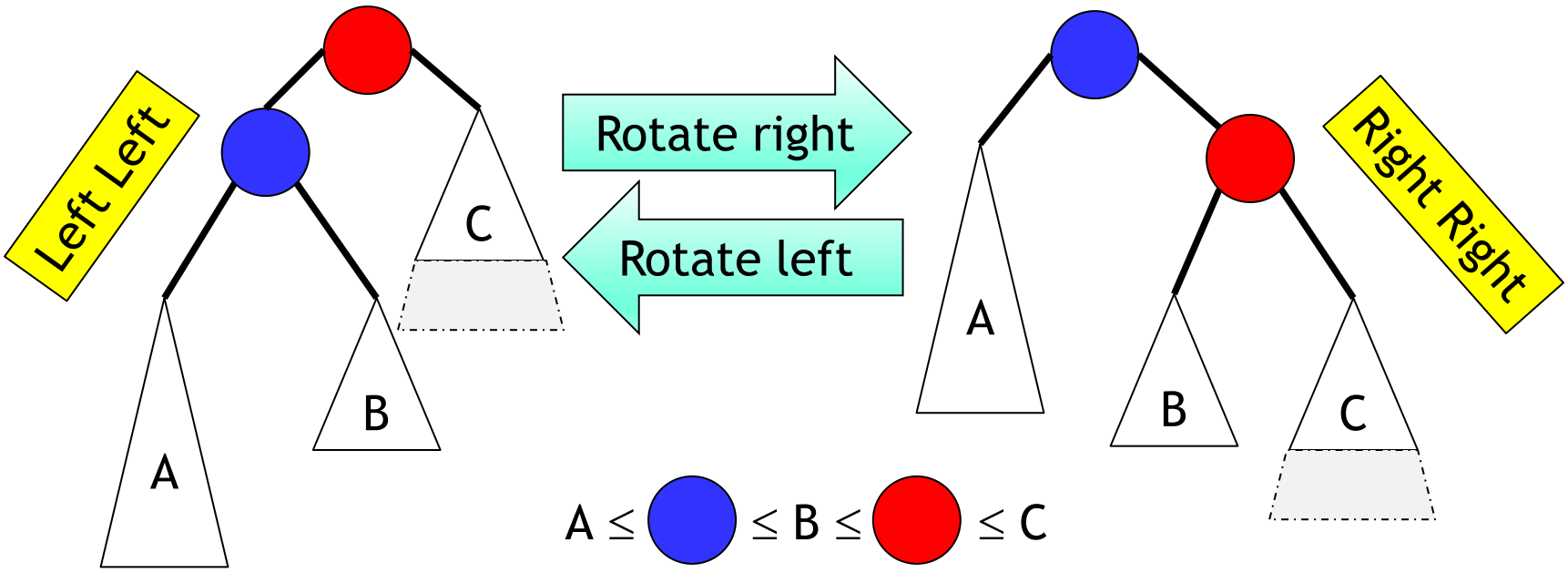
- No way to signal failure on empty queue

```
int peek(Queue q, int *peeked_value);

if(peek(myQueue, &first)) { ... }

int peek(Queue q, int *error_flag);

first = peek(myQueue, &ok);
if(ok) { ... }
```

Balanced Search Trees

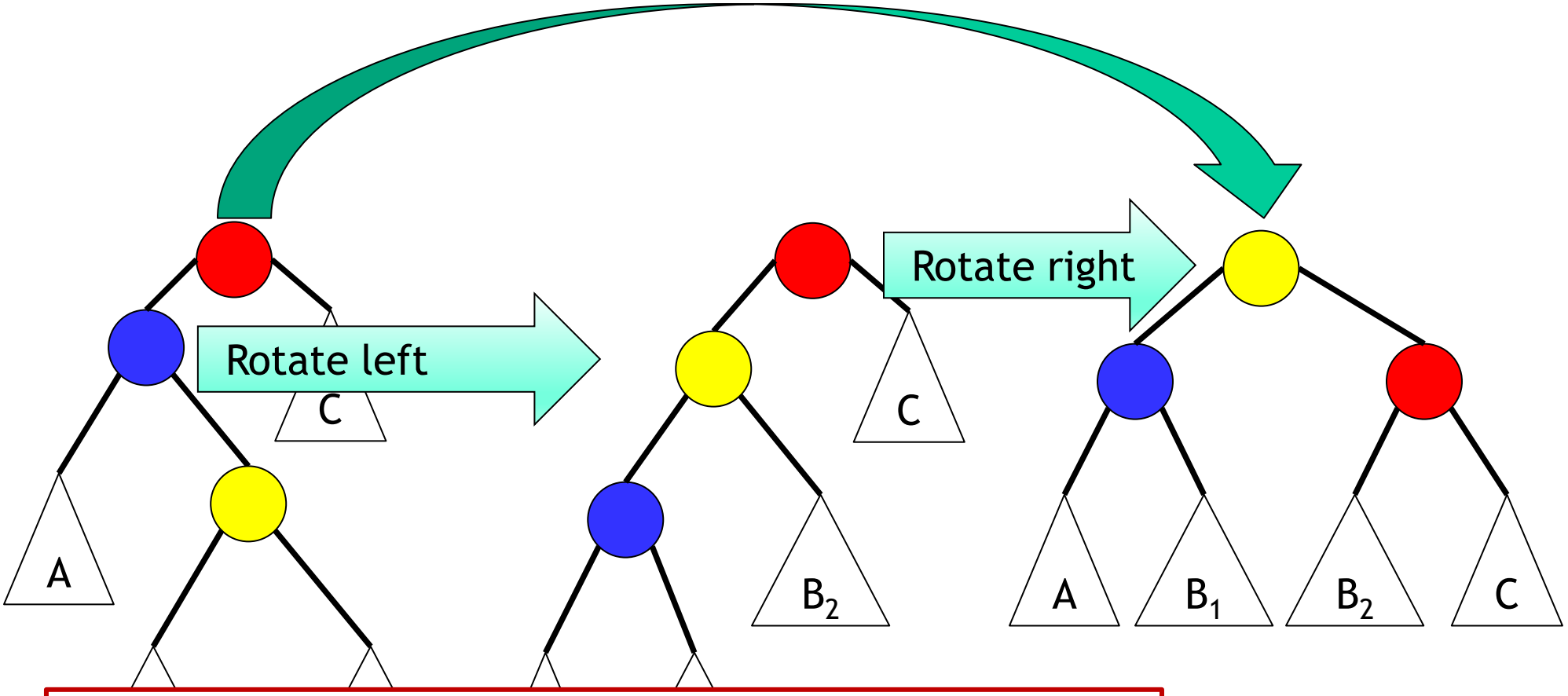
- How long does it take to determine whether a particular value *x* appears in an ordered tree *T* containing *n* nodes?
  - Worst case time is $O(h)$, where *h* is the height of *T*
  - Minimum height (*T* is balanced): $h = \lfloor \log_2 n \rfloor$
  - Maximum height (*T* is lopsided): $h = n - 1$

- *Key idea*: Keep the tree balanced (prevent it from getting lopsided)



Dealing with Left Right and Right Left

- Do a double-rotation!



[PDF] SOVIET MATHEMATICS DOKLADY
professor.ufabc.edu.br/~jesus.mena/courses/mc3305-2q.../AED2-10-avl-paper.pdf
by GM Adel'son-Vel'skii - 1962 - Cited by 4 - Related articles
An algorithm for the organization of information by G. M. Adel'son-Vel'skii and E. M. Landis.
Soviet Mathematics Doklady, 3, 1259-1263, 1962.

sure that height
tween left and
s is at most 1

AVL Tree Implementation

```
typedef struct node {
  int data; /* or some other type */
  struct node *left;
  struct node *right;
  int height;
} tNode;
```

```
int h(tNode *node) {
  return node == NULL ? -1 : node->height;
}
```

```
tNode *makeNode(int D, tNode *L, tNode *R) {
  tNode *node = (tNode *) malloc(sizeof(tNode));
  ...
  node->height = 1 + MAX(h(L), h(R));
  return node;
}
```

Insert in an AVL tree

```
tNode *insert(tNode *node, int D) {
  if(find(node, D)) { return NULL; }
  tNode *newNode = makeNode(D, NULL, NULL);
  if(newNode == NULL) { return NULL; }
  return rec_ins(node, newNode);
}

tNode *rec_ins(tNode *node, tNode *newNode) {
  if(node == NULL) { return newNode; }
  if(newNode->data < node->data) {
    node->left = rec_ins(node->left, newNode);
    if(h(node->left) > h(node->right)) {
      node->height = h(node->left) + 1;
    }
  } else if(newNode->data > node->data) {
    node->right = rec_ins(node->right, newNode);
    if(h(node->right) > h(node->left)) {
      node->height = h(node->right) + 1;
    }
  }
  return balance(node);
}
```

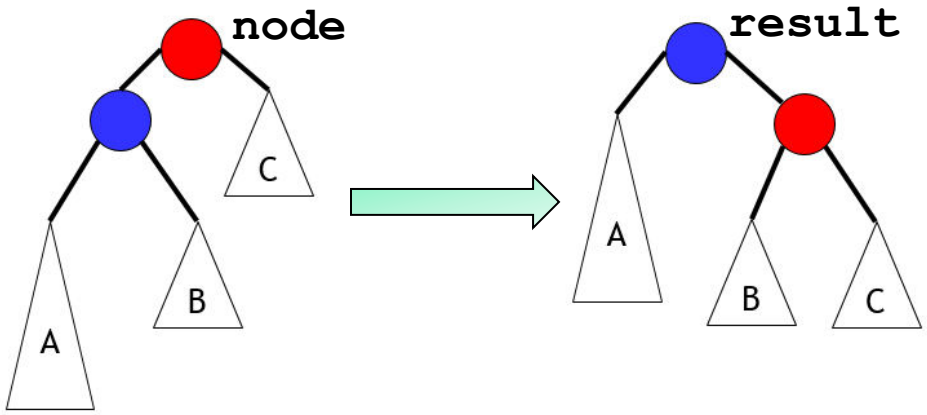
Balancing nodes in an AVL tree

```
tNode *balance(tNode *node) {
  if(node == NULL) { return NULL; }
  if(h(node->left) > h(node->right)+1) {
    if(h(node->left->left) > h(node->left->right) {
      return handleLL(node);
    } else {
      return handleLR(node);
    }
  } else if(h(node->right) > h(node->left)+1) {
    if(h(node->right->right) > h(node->right->left)) {
      return handleRR(node);
    } else {
      return handleRL(node);
    }
  } else { return node; }
}
```

handleLL

```
tNode *handleLL(tNode *node) {
  tNode *A, *B, *C, *result;
  result = node->left;
  A = result->left;
  B = result->right;
  C = node->right;
  result->right = node;
  node->left = B;
  node->height = 1 + MAX(h(B), h(C));
  result->height = 1 + MAX(h(A), node->height);
  return result;
}
```

O(1) time
Hence, balance() is also *O*(1) time
Thus, insert is *O*(*h*)
We will prove that *h* is *O*(log *n*)



Delete in an AVL tree

```
tNode *delete(tNode *node, int D) {
  if(node == NULL) { return NULL; }
  if(D < node->data)
    node->left = delete(node->left, D);
  else if(D > node->data)
    node->right = delete(node->right, D);
  else {
    if(node->left != NULL)
      node->left = extractMax(node->left, node);
    else if(node->right != NULL)
      node->right = extractMin(node->right, node);
    else { free(node); return NULL; }
  }
  node->height = 1 + MAX(h(node->left), h(node->right));
  return balance(node);
}
```

Space overhead per node

- Our AVL tree nodes store the (absolute) height of each node in the tree

```
typedef struct node {
  int data;
  struct node *left;
  struct node *right;
  int height;  // sizeof(int) overhead per node
} tNode;
```

- This can be reduced to two bits overhead per node:
  - LeftHeavy, Balanced, RightHeavy

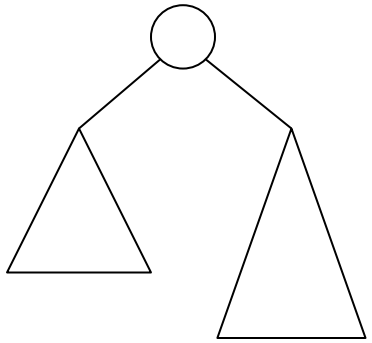
- Slightly more complex insert/delete/balance
  - another example of the space/time tradeoff!

AVL trees with n nodes have $O(\log n)$ height

- **Recall:** AVL Trees ensure that height difference between left and right sub-trees is at most 1

- Let $m(h)$ be the minimum number of nodes in an AVL tree of height h
  - If an AVL tree with n nodes has height h then $n \geq m(h)$
  - Hence, $h \leq m^{-1}(n)$ [assuming m is an increasing function]

- Now $m(0) = 1$, $m(1) = 2$, $m(2) = 4$, $m(3) = 1 + m(2) + m(1) = 7$



- $\forall h \geq 2,\ m(h) = 1 + m(h-1) + m(h-2)$

  $F(0) = F(1) = 1;\ \forall h \geq 2,\ F(h) = F(h-1) + F(h-2)$
  - Prove that $\forall h \geq 0,\ m(h) = F(h+2) - 1$

Critique this "proof"

- **Given**: $m(0) = 1$, $m(1) = 2$; $\forall h \geq 2$, $m(h) = 1 + m(h-1) + m(h-2)$
  and $F(0) = F(1) = 1$; $\forall h \geq 2$, $F(h) = F(h-1) + F(h-2)$

- We will prove that $\forall h \geq 0$, $m(h) = F(h+2) - 1$ by induction on h

- *Base case* ($h = 0$): $m(0) = 1$ and $F(2) = F(0) + F(1) = 1 + 1 = 2$
  — so the base case holds ☺

- *Inductive step* ($h > 0$): $m(h) = 1 + m(h-1) + m(h-2)$ Fails for $h = 1$
  $= 1 + F(h+1) - 1 + F(h) - 1$ by the inductive hypothesis
  $= F(h+2) - 1$ by the definition of F
  ☺

Fibonacci numbers and the Golden Ratio

- It turns out that

$$F(h) = \frac{\varphi^{h+1} - (1-\varphi)^{h+1}}{\sqrt{5}}$$

where $\varphi = \frac{1+\sqrt{5}}{2} = 1.618\cdots$ is the golden ratio

- Thus for large *h*, $m(h) = F(h+2) - 1 \geq \frac{1}{\sqrt{5}} 1.618^{h+3}$

- Thus, for an AVL tree with *n* nodes and height *h*

$n \geq \frac{1}{\sqrt{5}} 1.618^{h+3}$ and hence

$$h \leq \log_{1.618}\left(\sqrt{5}n\right) - 3$$
$$< \log_{1.618} n \approx 1.44 \log_2 n$$

- Thus, the height of an AVL tree can be about 44% more than optimal
  - find, insert and delete in *O*(log *n*) worst-case time

Dictionary ADT

- A dictionary stores (key, value) pairs where keys are totally ordered
  - e.g., integers, reals, vectors, strings, … `int cmp(key_t, key_t)`
  - For a key *k*, a dictionary has at most one pair (*k*, *v*)

```
#ifndef DICT_H
#define DICT_H
typedef void* DictEntry;
void setValue(DictEntry, value_t);
value_t getValue(DictEntry);

typedef void* Dict;
void freeDict(Dict);
int isEmpty(Dict);
int size(Dict);
Dict insert(Dict, key_t, value_t);
DictEntry get(Dict, key_t);
Dict delete(Dict, key_t);
DictEntry forall(Dict, int(*task)(int, void*), void*);
#endif
```

Assuming cmp runs in $O(1)$ time!

Linked list	AVL tree
$O(1)$	$O(1)$
$O(n)$	$O(n)$
$O(n)$	$O(\log n)$

Replace old **`value`** if **`key`** already exists

Research Note

- A comparison of dictionary implementations by Mark Neyer, 2009
- This paper explores several implementations of dictionaries:
  - hash tables, Red-Black Trees, AVL Trees, and Skip Lists

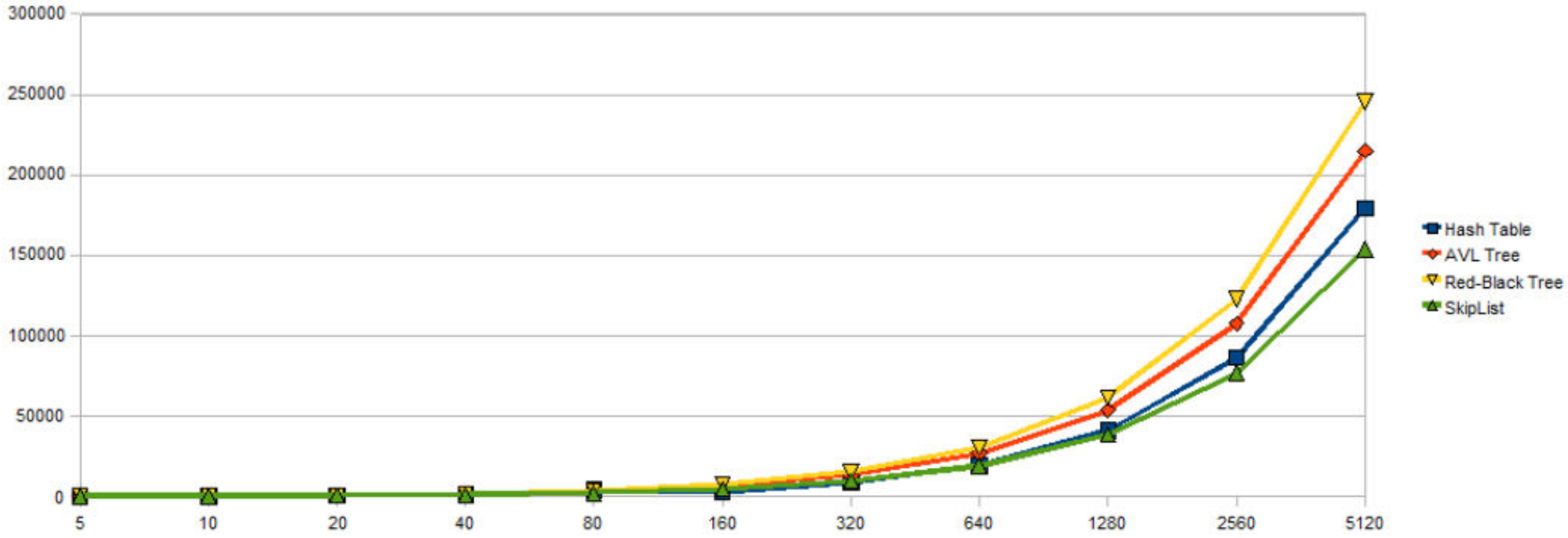
- The Microsoft .NET platform was chosen for this experiment because it is of interest to the author's experience in industry, and due to the availability of a precise memory measurement debugging library, namely **`sos.dll`**

- Implementations
  - System.Collections.hashtable
  - AVL tree implementation: www.vcskicks.com
  - Red-Black tree implementation: www.devx.com/DevX/Article/36196
  - Skip List implementation: www.codeproject.com/KB/recipes/skiplist1.aspx

No mention of hardware!

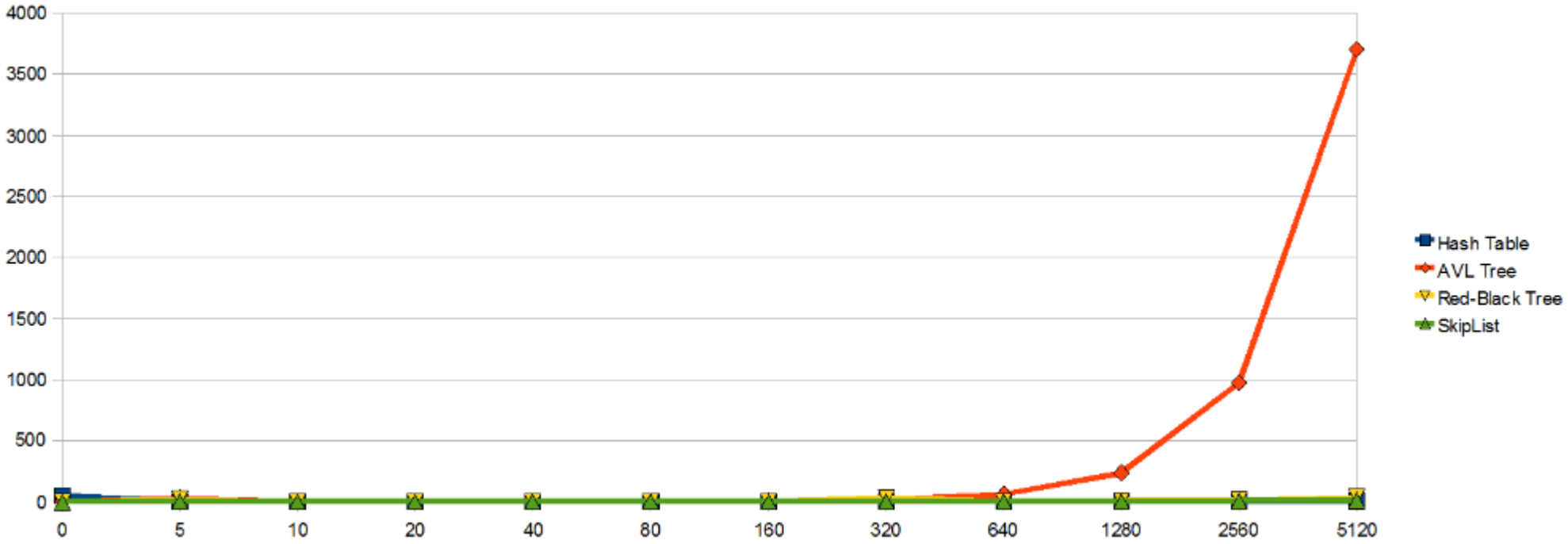
Results (1/3)

- Size (in bytes) vs. *n*



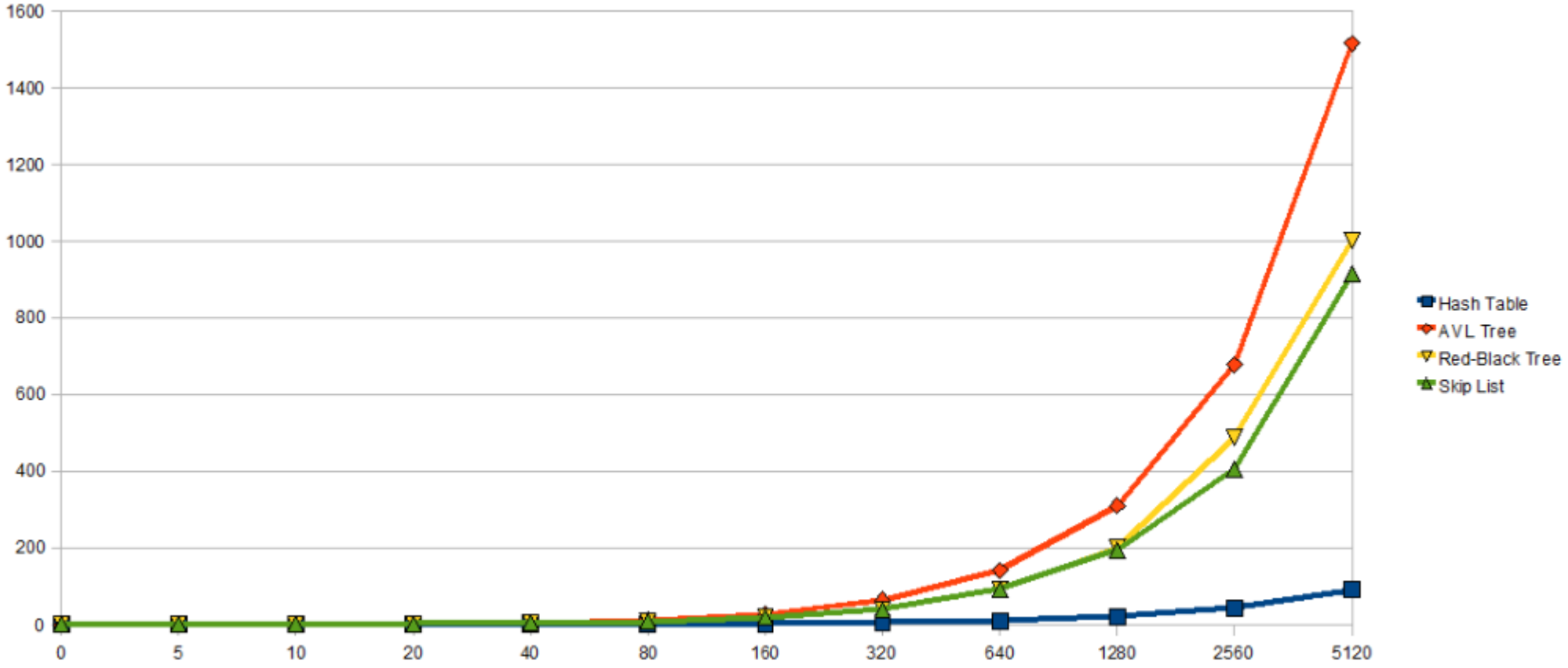
Results (2/3)

- Creation time (in milliseconds) vs. *n*



Results (3/3)

- Query time for 100*n* keys (in milliseconds) vs. *n*



Priority Queues

- A priority queue is an ADT for storing (priority, data) pairs, where priorities are totally ordered (e.g., integers, strings, etc.)
  - Two variants (silly English):
    - low-magnitude priority means “high priority”
    - high-magnitude priority means “high priority”

```
#ifndef PQUEUE_H
#define PQUEUE_H
typedef void* PQEntry;
typedef void* PQueue;
int size(PQueue);
PQEntry insert(PQueue *pq, int priority, int data);
int deleteNext(PQueue pq, int *priority, int *data);
PQEntry update(PQueue pq, PQEntry e, int newPriority);
#endif
```

Priority Queue Implementations

	Linked List	Ordered Linked List	Ordered Binary Tree	Balanced Ordered Binary Tree
size	$O(1)$	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$
deleteNext	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$
update	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$

- **Interview question**: All n (priority, data) items known in advance, only need deleteNext. What is the best implementation?

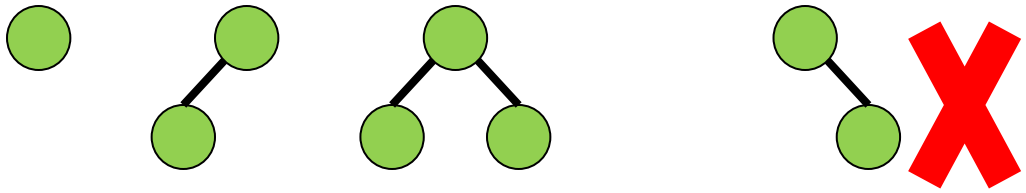
- AVL trees: pointer-based, large constant hidden in the $O(.)$

- We will develop a fast, array-based implementation
  - size in $O(1)$ time
  - insert and deleteNext in $O(\log n)$ time [smaller constant]

Complete Binary Trees

- A binary tree is complete if all levels (except possibly the last) are completely filled, and all nodes are as far left as possible

- *Examples*:



- **Fact**: A complete binary tree with n nodes has height exactly $\lfloor \log_2 n \rfloor$

- Array representation:
  - root at index 0
  - for any node at index i:
    - left child at index $2i + 1$
    - right child at index $2i + 2$
    - parent at index $\lfloor (i - 1)/2 \rfloor$ (unless $i = 0$)

Heaps

- **Definition:**
  - A minheap is a complete binary tree in which every node's priority is less than or equal to the priority of its children (min. at root)
  - A maxheap is a complete binary tree in which every node's priority is greater than or equal to the priority of its children (max. at root)

- We implement a heap as an array

- **Question:** Which of these arrays represent minheaps?

A	1	2	3	4	5	6

B	1	2	4	5	6	3

C	1	3	4	2	6	5

D	1	4	2	6	5	3

Sift Up

- Suppose the heap is implemented as a **`struct`** with array **`a[]`** where each array is a **`struct`** with priority **`p`** and data **`d`**

```
void siftUp(Heap *H, int i) {  // minheap
  while(i > 0) {
    int j = (i-1)/2;  // parent
    if(H->a[i].p >= H->a[j].p) { return; }
    SWAP(H->a, i, j);
    i = j;
  }
}
```

Why not: **`SWAP(H->a[i], H->a[j])`**
Temporary?

- What does **`siftUp(H, 3)`** do if **`H->a`** =

1	3	4	2	6	5

- Insert = append to array + sift up the last element: $O(\log n)$ time

Sift Up

- Suppose the heap is implemented as a **`struct`** with array **`a[]`** where each array is a **`struct`** with priority **`p`** and data **`d`**

```
void siftUp(Heap *H, int i) {  // minheap
  while(i > 0) {
    int j = (i-1)/2;  // parent
    if(H->a[i].p >= H->a[j].p) { return; }
    SWAP(H->a, i, j);
    i = j;
  }
}
```

- Insert = append to array + sift up the last element: $O(\log n)$ time

- Time to "heapify" n items: $O(n \log n)$

Sift Down

```
void siftDown(Heap *H, int i) {  // minheap
  while(i < (H->size)/2) {  // i >= size/2 is leaf!
    int j = 2*i+1;  // left child
    if(j < H->size-1 && H->a[j].p > H->a[j+1].p)
      j++;  // right child
    if(H->a[i].p <= H->a[j].p) { return; }
    SWAP(H->a, i, j);
    i = j;
  }
}
```

- What does `siftDown(H, 0)` do if `H->a` = | 7 | 2 | 4 | 3 | 6 | 5 |
- deleteNext = swap root with last, `size--`, siftDown(0): $O(\log n)$ time
  - update?

Heapify in $O(n)$ time

```
void heapify(Heap *H) {  // minheap
  for(int i = H->size/2-1; i >= 0; i--) {
    siftDown(H, i);
  }
}
```

Sloppy

- Runtime analysis: $O(n)$ iterations, each takes $O(\log n)$ time: $O(n \log n)$
- Careful runtime analysis?
- Heapsort
- Two advanced data structures: Treaps and Fibonacci heaps

B-trees

Organization and maint

R Bayer, E McCreight - Softwa

Summary Organization and m
considered. It is assumed that
backup store like a disc or a d

Cited by 2

BOEING SCIENTIFIC RESEARCH LABORATORIES DOCUMENT D1-82-0889

BOEING

SCIENTIFIC RESEARCH LABORATORIES

- Actually

Organizat
access file is
on some pseudo
index organi
of keys in tim
index and k
formance of th
least 50% but
in a special d
performance bo

References:

Adelson-Velskii, G. M. and Landis, E. M. An Information Organization Algorithm. DANSSSR, No. 2, 1962.

Foster, C. C. Information Storage and Retrieval Using AVL Trees. *Proc. ACM 20th Nat'l. Conf.* (1965), pp. 192-205.

Gladun, V. P. Storage Organization for Key Search and Recording. *Cybernetics*, Vol. 1, No. 4, August 1965.

Landauer, W. I. The Balanced Tree and Its Utilization in Information Retrieval. *IEEE Trans. on Electronic Computers*, Vol. EC-12, No. 6, December 1963.

Sussenguth, E. H., Jr. The Use of Tree Structures for Processing Files. *Comm. ACM*, Vol. 6, No. 5, May 1963.

Large amount of data

- Need to store most of the data on disk (or in the cloud)
  - getting data from these sources is MUCH slower than RAM

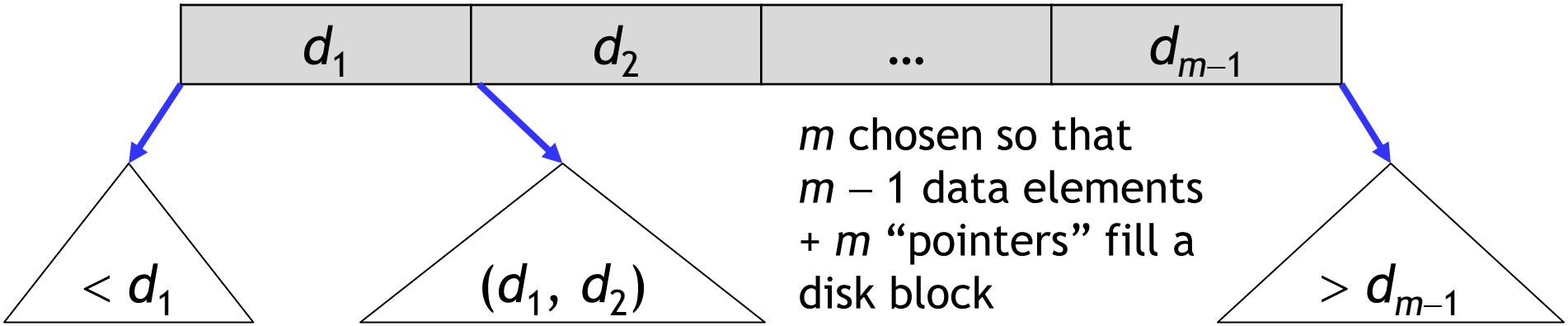
Data from 2014

Component	Latency
1 CPU cycle	0.3 ns
L1 cache access	0.9 ns
L2 cache access	2.8 ns
L3 cache access	12.9 ns
RAM access	120 ns
SSD access	50-150 μs
HDD access	1-10 ms
SF to NYC internet	40 ms

- **Key idea**: If we have to go to the disk, make it count!
  - each disk access fetches a whole block of data (a few KB)

Nodes in a B-tree

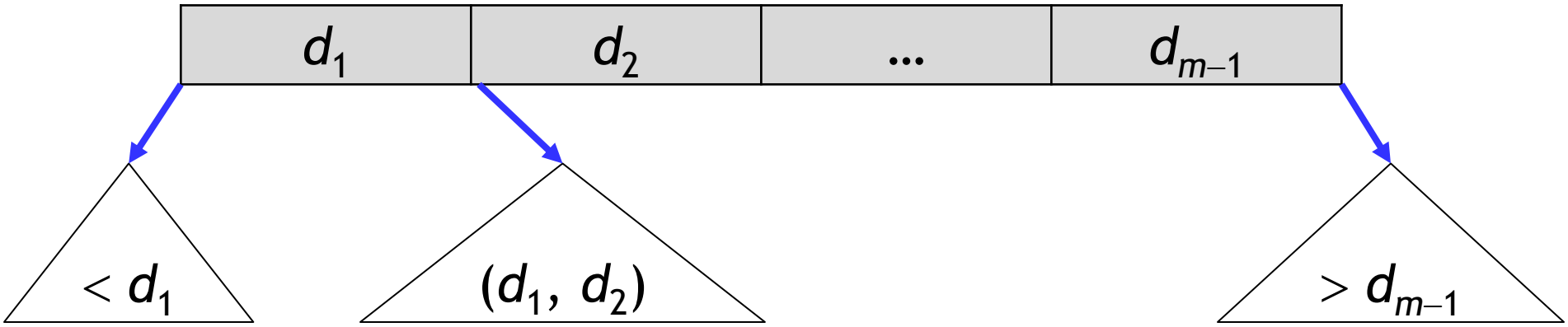
- In a B-tree of order m, each node has up to m children



- The root is either a leaf, or has at least two children
- Each internal node except the root has between $\lceil m/2 \rceil$ and m children
- All leaves are at the same level
  - tree height is $\approx \log_m n$
  - m is of the order of 10^2 or more, so for $n \approx 10^9$, depth is around 5

Operations on B-trees

- Find is a generalization of Find for ordered binary trees
  - binary search used to find the correct pointer to follow



- To insert *d*, find the leaf node where it should have been
  - insert into that node (in sorted order)
  - if the node was full (i.e., it had $m - 1$ elements), promote the middle element to the parent-level and split this into two half-filled nodes
  - promote/split may cascade up to the root (tree height will grow)

- Delete is similar: remove data, then try to compress with adjacent sibling

An optimization: B^+ trees

- Suppose data is made up of (small) keys and (large) values
  - search done on the basis of keys
- Internal nodes only store keys
  - values are stored only at leaves
- Thus, a single disk block can accommodate many more keys
  - bushier tree $\Rightarrow$ shallower tree
- Insert animation from OpenDSA

B/B$^+$ trees: Frequent nodes

- In both B-trees and B$^+$-trees, the root is processed frequently
- While the data structure is needed, the root can be stored in RAM
  - the root can be written out to the disk when the data structure is no longer needed by the program (e.g., when the program terminates)
- RAM is large enough to accommodate more than just the root
  - but it cannot accommodate everything
- *Note*: This is exactly the problem of caching
  - cache faster than RAM (just like RAM faster than disk)
  - cache smaller than RAM (just like RAM smaller than disk)
- **Key Question**: What to save and what to evict?

Eviction Strategies

- FIFO: First In First Out
  - data structure?
- LRU: Least Recently Used
  - data structure?
- FF: Furthest in Future
  - provably optimal
  - data structure?
- Aside: Splay trees
  - whenever you insert or find something, splay it to the top
  - $O(\log n)$ amortized find/insert/delete

Questions

1. In a full tree, every node has either 0 or 2 children
   - What is the maximum height of a full tree with *n* nodes

2. Given pointers *p* and *q* to two nodes in an balanced binary search tree (ordered binary tree) with *n* nodes, describe a fast algorithm to find the lowest common ancestor of *p* and *q*
   - Answer the same question, assuming that the tree is unordered and not necessarily balanced

3. In a general tree, nodes can have an arbitrary number of children
   - Describe a space-efficient way to implement general trees in C