Binary search trees

Binary search
Binary search tree operations

Read pages 1367-1386 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

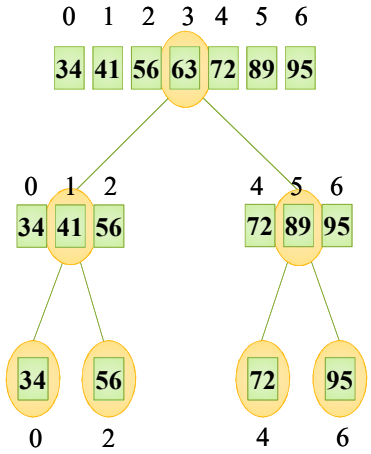
Binary search

- Is an example of divide and conquer
- Assumes that the data to be searched is arranged in order
  - Either smallest to largest or largest to smallest
- Finds a required item much quicker than a linear search
- Assuming the items are ordered and stored in an array:
  - Check to see whether the item in the middle of the array is the one required
    - If **yes**, return the index of the item
    - If **no**, repeat this algorithm on either the left or right sub-arrays
    - If the sub-array is empty, return some indicator that the required item is not present (e.g. return –1)

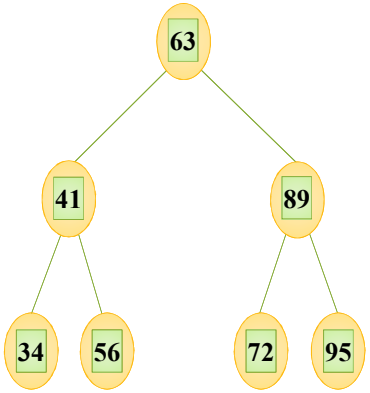
Binary search algorithm

```
// Finds a target value in an array of integers.
// Initially, first = 0 and last = a.size()-1
int find(int[] a, int target, int first, int last){
   if (first > last)
   {
      return -1; // target not there
   }
   else
   {
      // compute midpoint
      int mid = (first + last) / 2;

      if (a[mid] == target) {return mid;} // target found
      else // search left half
      if (a[mid] > target) {return find(a, target, first, mid-1);}
      else // search right half
      {return find(a, target, mid+1, last);}

   }
}
```

Binary search calls



Data arranged in a binary search tree



Binary search trees

- Each node of a binary search tree is
  - Bigger than its left child, and
  - Less than or equal to its right child (* this has to be documented)
- Therefore, for any node:
  - Each item in the left sub-tree is less than the item in the parent
  - Each item in the right sub-tree is bigger than or equal to the item in the parent
- Searching these trees is most efficient if the tree is balanced
  - That is its depth is kept to a minimum

Pseudocode for recursive search

```
Item find(node, target)

   if the node is null
      return null

   else if the item equals the target
      return the item

   else if the target is less than the item
      return find(node.left, target)

   else
      return find(node.right, target)
```

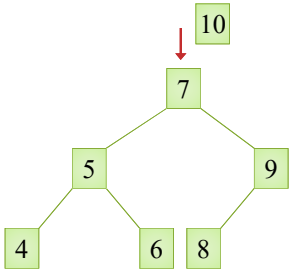
Inserting into a binary search tree

- If the tree already contains an item equal to the inserted item, add a copy. If the data structure should not allow copies, then the existing item should be replaced.
  - Items to be inserted must be comparable, and required operators need to be overloaded
- Otherwise insert the item into its correct place in the tree, for example:

```
BSTree* tree = new BSTree(); // creates a new tree

// adds content
tree->add("Bill");
tree->add("Mary");
tree->add("John");
tree->add("Alex");
```

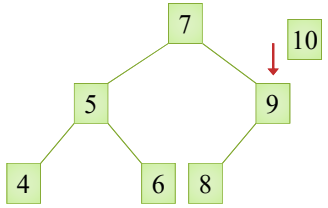
Inserting into a binary search tree

- First case: The tree is empty
  - Set the root to a new node and store the item at that node
- Second case: The tree already contains one or more items
  - Call a recursive helper method to insert the item



Inserting into a binary search tree

- First case: The tree is empty
  - Set the root to a new node and store the item at that node
- Second case: The tree already contains one or more items
  - Call a recursive helper method to insert the item



Inserting into a binary search tree

```
// Recursive method for
// insertion into a binary tree

void insert(Item it)
{
   if (root == NULL)
      {root = new BTNode(it);}
   else
      {add(root, it);}

   size++;
}
```

```
// This is the method for the private helper

add(BTNode node, Item it)
{
   if (it == node.value)
   {
      node.value = it; // Replace existing item
   }

   else if (it > node.value)
   {  // Work on right side
      if (node.right == NULL)
         node.right = new BTNode (it);
      else
         add(node.right, it);
   }

   else
   {  // Work on left side
      if (node.left == NULL)
         node.left = new BTNode (it);
      else
         add(node.left, it);
   }
}
```

Finding an item

```
// Pseudocode for search()
Item search(Item target)
{
   return find(root, target);
}


// this is the recursive helper method
Item find (BTNode node, Item target)
{
   if (node == NULL) {return NULL;}
   else
   {
      if (it == node.value)
      {
         return node.value;
      }
      else if (it > node.value)
      {
         return find(node.right, target);
      }
      else {return find(node.left, target);}
   }
}
```

Printing out a binary tree

```
// Pseudocode for inorder toString()
string toString()
{
   return toString (root, 0);
}


string toString (BTNode node, int level)
{
   string str = "";
   if (node != NULL)
   {
      // Right sub-tree
      str += toString (node.right, level + 1);

      // Indent to indicate level
      for (int i = 1; i <= level; ++i) {str = str + "| ";}

      // Current node
      str += node.value.toString() + "\n";

      // Left sub-tree
      str += toString (node.left, level + 1);
   }
   return str;
}
```

DEMO

Effect of order of insertion

- The order in which items are inserted can have a dramatic impact on the shape and performance of a binary search tree
  - The tree can vary from well balanced to linear
  - Performance degrades as the tree varies from well balanced
  - It is possible to construct trees that retain a good balance, no matter what the insertion order

Perfect insertion order

```
BSTree* tree = new BSTree();
cout << ">>>>Items in advantageous order:";

tree->insert ("E");
tree->insert ("C");
tree->insert ("D");
tree->insert ("A");
tree->insert ("H");
tree->insert ("F");
tree->insert ("K");
tree->toString();
```

```
>>>>Items in advantageous order:
| | K
| H
| | F
E
| | D
| C
| | A
```

Pathologically bad insertion order

```
cout << ">>>>Items in the worst order:";
tree = new BSTree();
for (int i = 1; i <= 8; i++)
   tree->insert("" + i);
tree->toString();
```

```
>>>>Items in worst order:
| | | | | | | 8
| | | | | | 7
| | | | | 6
| | | | 5
| | | 4
| | 3
| 2
1
```

Random insertion order

```
cout << ">>>>Items in random order:");
tree = new BSTree();
for (int i = 1; i <= 8; i++)
// Uses a function that randomly
// generates single character
// Strings
tree->insert(randomString());
tree->toString();
```

```
>>>>Items in random order:
¦ X
¦ ¦ U
P
¦ ¦ O
¦ H
¦ ¦ F
¦ ¦ ¦ B
```

Two variations of binary tree implementations

- First type: There are `BTree` and a `BTNode` classes and each node itself is a tree with a root, linked to the neighbour nodes. You build the tree by connecting trees to other trees.
- Second type: There is only one `BTree` instance, that points to the `root` node. From the `root` node you can move to any other node using a `current` pointer. You move `current` around to build the tree.
- The two types are very similar, but there are differences on how you populate the tree. The implementation of some methods will also change. Both are 'correct'.

DEMO

Hash Tables

Hash tables fundamentals

The textbook does not cover hash tables, but you can get find plenty of information online:

https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/

What are hash tables? Why use them?

- Hash tables offer the possibility of constant time search, insertion and removal, by mapping elements to positions through the use of a hashing function, which is fast to calculate.
- There are limitations to the approach and issues that might arise depending on the nature of the data to be stored.
- There are solutions for those issues, but those can be computationally costly.
- Implementing good hash-based data structures is a complex matter and there are no simple recipes. Performance is highly dependent on the compatibility between the data to be stored and the characteristics of the hash table.

Hash function

- Hash tables rely on a hash function, which maps the elements to be stored, to a position in the data structure.
- This mapping is calculated using a mathematical formula.
- Ideally, each item should have a unique has value.
- If two different items map to the same position in the data structure, a collision will happen. Fixing that collision will require additional calculations, which will increase the computational cost, and reduce the efficiency of the data structure.
- Calculation of the hash function is performed on each insertion, access and removal.

Data structure example

- Consider a hash table that uses an array as the underlying data structure and consider that strings are to be stored.
- The index of the array cell in which the item is stored is calculated from the result of applying the hash function to the item
  - This maintains constant access time
- Suppose we have an array with 15 positions, and we want to store the following strings:

```
alex, peter, mary, john, lauren, monica, frank, bruce
```

Data structure example

- Moreover, consider that the hash function is

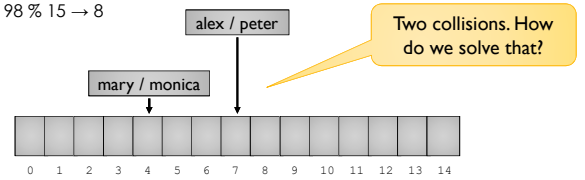
```
int hashFunction (string value)
{
    return ((int)value.at(0)%15);
}
```

- The method `at(i)` returns the char at position `i` of the string.
- The `(int)` casting converts the char to its ascii code value.
- `%` returns the remainder of the division between `value.at(0)` and `15`.

Data structure example

- The ascii table is:
- And the mapping is:
  - alex → 97 % 15 → **7**
  - peter → 112 % 15 → **7**
  - mary → 109 % 15 → **4**
  - john → 106 % 15 → 1
  - lauren → 108 % 15 → 3
  - monica → 109 % 15 → **4**
  - frank → 102 % 15 → 12
  - bruce → 98 % 15 → 8

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]



Data structure example (2)

- Let's change the hash function:

```
int hashFunction (string value)
{
    return (((int)(value.at(0)+(int)(value.at(1)))%15);
}
```

- The function adds the two first characters' ascii codes.
- What does it look like?

Data structure example (2)

- The ascii table is:
- And the mapping is:
  - alex → (97+108) % 15 → **10**
  - peter → (112+101) % 15 → 3
  - mary → (109+97) % 15 → 11
  - john → (106+127) % 15 → 8
  - lauren → (108+97) % 15 → **10**
  - monica → (109+111) % 15 → **10**
  - frank → (102+114) % 15 → 6
  - bruce → (98+114) % 15 → 2

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

Two collisions again... Now what?

Increase the size of the array! Let's try 30!

Data structure example (3)

- The ascii table is:
- And the mapping is:
  - alex → (97+108) % 30 → **25**
  - peter → (112+101) % 30 → 3
  - mary → (109+97) % 30 → 26
  - john → (106+127) % 30 → 23
  - lauren → (108+97) % 30 → **25**
  - monica → (109+111) % 30 → 10
  - frank → (102+114) % 30 → 6
  - bruce → (98+114) % 30 → 2

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

One collision. The problem is that "alex" and "lauren" have the same hash value.

Data structure example (3)

- Let's change the hash function again:

```
int hashFunction (string value)
{
    return (((int)(value.at(0) + 3*(int)(value.at(1)))%15);
}
```

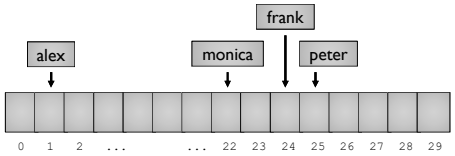
- The function adds the two first characters' ascii codes, but the second char is multiplied by 3, so "al" ≠ "la".
- What does it look like?

Data structure example (3)

- The ascii table is:
- And the mapping is:
  - alex → (97+324) % 30 → 1
  - peter → (112+303) % 30 → 25
  - mary → (109+291) % 30 → 10
  - john → (106+381) % 30 → 7
  - lauren → (108+291) % 30 → 9
  - monica → (109+333) % 30 → 22
  - frank → (102+342) % 30 → 24
  - bruce → (98+342) % 30 → 20

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

No collisions!



Hash tables

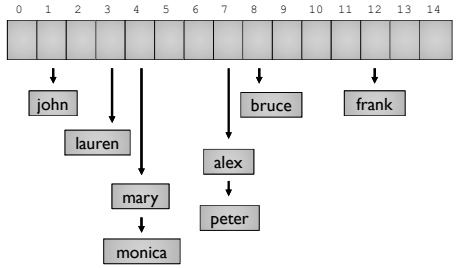
- Now, how can you avoid collisions?
- Increase the size of the underlying array, but that costs memory.
- Resize the array as needed.
- For instance, the last example is OK, but if we have to store 20+ names, it is likely that we will get a collision. In that case, we can resize the array from 30 to 60, and "re-hash" all the elements currently stored.
- Of course, that will cost time. If you have to do that too frequently, it defeats the purpose of having a hash table.
- Designing a hash table requires deep understanding of the data to be stored, and of how hash functions work.

More on collisions - Chaining

- What if the underlying data structure is not an array, but something more complex?
- What if each position of the array points to a linked list that stores the elements?

Data structure example

- The first mapping that we had becomes
  - alex → 97 % 15 → 7
  - peter → 112 % 15 → 7
  - mary → 109 % 15 → 4
  - john → 106 % 15 → 1
  - lauren → 108 % 15 → 3
  - monica → 109 % 15 → 4
  - frank → 102 % 15 → 12
  - bruce → 98 % 15 → 8



- Searching for an element involves finding the correct position in the array (the correct "bucket"), and then a linear search on the linked list attached to that bucket.

More on collisions

- What if the underlying data structure is not an array, but something more complex?
- What if each position of the array points to a linked list that stores the elements?
- What if each position points to a binary search tree?
  - That would be very efficient.
  - There would be a constant time calculation to find the correct bucket.
  - And a binary search to find the correct element.
  - But it is more difficult to implement.

See you next week!