CSE 12 Discussion 9

Max Jiao & Pooja Bhat

Review Quiz

Quadratic Hashing and Separate chaining both solve the problem of primary clustering in the table

A. True for both
B. True for Quadratic Hashing, False for Separate Chaining
C. False for Quadratic Hashing, True for Separate Chaining
D. False for both

Review Quiz

Quadratic Hashing and Separate chaining both solve the problem of primary clustering in the table

A. True for both
B. True for Quadratic Hashing, False for Separate Chaining
C. **False for Quadratic Hashing, True for Separate Chaining**
D. False for both

Review Quiz

When the _____ is too high, we need to _______

A. Load factor, rehash
B. Collision rate, double hash
C. Hashing rate, double hash
D. Table size, create another table
E. Probe length, separate chaining

Review Quiz

When the _____ is too high, we need to _______

A. **Load factor, rehash**
B. Collision rate, double hash
C. Hashing rate, double hash
D. Table size, create another table
E. Probe length, separate chaining

Review Quiz

Why do we have to resize the table before it gets completely full?

A. Because space complexity is cheap
B. Because it amortizes time
C. Because at some point too many collisions occur
D. This is just convention

Review Quiz

Why do we have to resize the table before it gets completely full?

A. Because space complexity is cheap
B. Because it amortizes time
C. **Because at some point too many collisions occur**
D. This is just convention

Review Quiz

Given an empty table of size 11, named hasher, that uses the Separate Chaining technique for hashing.

the hash function is:
h = (k*2 + 1) % size

How would the table look after the following calls?

hasher.add(19);
hasher.add(1);
hasher.add(3)
hasher.add(19);
hasher.add(7);

(Use the format idx0: 3 means value 3 at index 0 in the table)

A. Error: There is a duplicate!
B. Idx3: 1; idx4: 7; idx6: 19-->19; idx7: 3
C. Idx3: 1; idx4: 7; idx6: 19; idx7: 19; idx8: 3
D. Error: needs a second hash function for separate chaining
E. None of these

Review Quiz

Given an empty table of size 11, named hasher, that uses the Separate Chaining technique for hashing.

the hash function is:
h = (k*2 + 1) % size

How would the table look after the following calls?

hasher.add(19);
hasher.add(1);
hasher.add(3)
hasher.add(19);
hasher.add(7);

(Use the format idx0: 3 means value 3 at index 0 in the table)

A. Error: There is a duplicate!
B. Idx3: 1; idx4: 7; idx6: 19-->19; idx7: 3
C. Idx3: 1; idx4: 7; idx6: 19; idx7: 19; idx8: 3
D. Error: needs a second hash function for separate chaining
E. **None of these**

Review Quiz

A BST is similar to a hashtable in terms of uses and functionality (they share many of the same method calls, think back to hw7 search engine)

What are some of the pros and cons of each?

- BST has a slower runtime for all operations, but preserves order (ie has a way to print elements in order)
- Hashtables allow duplicate keys to be inserted
- Hashtables use contiguous memory, so it could potentially be problematic if the table gets too big
- BSTs are memory efficient, while hashtables are not
- Hashtables have a better best and worst case runtime for add/find

Review Quiz

A BST is similar to a hashtable in terms of uses and functionality (they share many of the same method calls, think back to hw7 search engine)

What are some of the pros and cons of each?

- **BST has a slower runtime for all operations, but preserves order (ie has a way to print elements in order)**
- Hashtables allow duplicate keys to be inserted
- **Hashtables use contiguous memory, so it could potentially be problematic if the table gets too big**
- **BSTs are memory efficient, while hashtables are not**
- Hashtables have a better best and worst case runtime for add/find

Course and TA evaluations are out

Please spend a few minutes to give feedback :)

Pointers

```
int *ptr;
int val = 1;
ptr = &val;
printf("%d", *ptr);       //      1.  What is the output?
printf("%d", *(&val));    //     2. What is the output?
```

A. 1, 1
B. Segmentation fault):
C. 1, unknown garbage value
D. Garbage value , 1

Pointers

```
int *ptr;
int val = 1;
ptr = &val;
printf("%d", *ptr);       //      1.  What is the output?
printf("%d", *(&val));    //     2. What is the output?
```

A. 1, 1

B. Segmentation fault):

C. 1, unknown garbage value

D. Garbage value , 1

Value at (the address of val)

Pointers

```
int *ptr;
int val = 1;
ptr = &val;
int val2 = 2;
ptr = &val2;
printf("%d", *ptr);     //  What is the output?
(*ptr)++;
printf("%d", *ptr);     //  What is the output?
printf("%d", val2);    // What is the output
printf("%d", val);     // What is the output
```

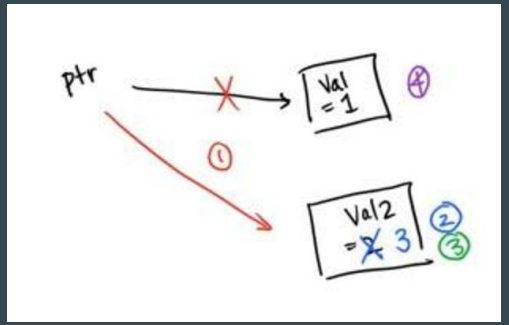
A. 1, 2, 2, 1
B. 2, 3, 2, 1
C. 2, 3, 3, 1
D. 2, 3, 2, 2
E. 1, 2, 1, 1

Pointers

```
int *ptr;
int val = 1;
ptr = &val;
int val2 = 2;
ptr = &val2;
printf("%d", *ptr);     //  1. What is the output?
(*ptr)++;
printf("%d", *ptr);     // 2.  What is the output?
printf("%d", val2);    // 3. What is the output
printf("%d", val);     // 4. What is the output
```

A. 1, 2, 2, 1
B. 2, 3, 2, 1
C. **2, 3, 3, 1**
D. 2, 3, 2, 2
E. 1, 2, 1, 1

Draw Memory diagram!



2. What is the output of the following code?

```
int main(int argc, char * argv[]) {
{
        int *ptr, a = 10;
        ptr = &a;
        *ptr += 2;
        printf("%d,%d\n", *ptr, a);
}
```

a) 10, 10
b) 10, 12
c) 12, 10
d) 12, 12

2. What is the output of the following code?

```
int main(int argc, char * argv[]) {
{
        int *ptr, a = 10;
        ptr = &a;
        *ptr += 2;
        printf("%d,%d\n", *ptr, a);
}
```

a) 10, 10
b) 10, 12
c) 12, 10
d) 12, 12

Remember that *pointer is getting the value a, which is stored in the memory location &a.

*ptr and a will share the same value!

Maybe they should have used @ instead of & to get address...
Too bad we're stuck with &

8. What is the output of the following code? Assume that the variable x is stored at memory address 1000.

```
int main(int argc, char * argv[]) {
 {

        int x = 0;
        int *ptr = &x;
        printf("%p, ", ptr);
        x++;
        printf("%p", ptr);
 }
```

a) 0, 1 b) 0, 0 c) 1000, 1001 d) 1000, 1000 e)1000, 1004

8. What is the output of the following code? Assume that the variable x is stored at memory address 1000.

```
int main(int argc, char * argv[]) {
 {

        int x = 0;
        int *ptr = &x;
        printf("%p, ", ptr);
        x++;
        printf("%p", ptr);
 }
```

a) 0, 1 b) 0, 0 c) 1000, 1001 **d) 1000, 1000** e)1000, 1004

(Value is incremented but address doesn't change)

Pointer Rules Summary

- A pointer may only be dereferenced after it has been assigned to refer to a pointee. Most pointer bugs involve violating this

- Allocating a pointer does not automatically assign it to refer to a pointee.
  In fact, every pointer starts out with a bad value. Correct code overwrites the bad value with a correct reference to a pointee, and thereafter the pointer works fine. There is nothing automatic that gives a pointer a valid pointee. (Compare it to Java where everything was defaulted to NULL)

- Assignment between two pointers makes them refer to the same pointee which introduces sharing.

Pointers

```
int *ptr1;
int val = 1;
ptr1 = &val;
int* ptr2 = ptr1;
int* ptr3 = &val;
 (*ptr2)++;
```

What are the values of val, *ptr1, *ptr2 and *ptr3?

A. 1, 1, 2, 1
B. 1, 2, 2, 2
C. 1, 2, 2, 1
D. 2, 2, 2, 2
E. 2, 2, 1, 1

Pointers

```
int *ptr1;
int val = 1;
ptr1 = &val;
int* ptr2 = ptr1;
int* ptr3 = &val;
 (*ptr2)++;       //note this is different from *(ptr2++)
```

What are the values of val, *ptr1, *ptr2 and *ptr3?

A. 1, 1, 2, 1
B. 1, 2, 2, 2
C. 1, 2, 2, 1
D. 2, 2, 2, 2
E. 2, 2, 1, 1

pointer sharing!

(think pass by reference in java)

Pointer and Arrays

```
int my_array[ ] = {1, 2, 3, 4, 5, 6};
int *ptr;
```

Which of the following assignment is correct?

a) ptr = &my_array[0];
b) ptr = my_array;
c) ptr = my_array[0];
d) ptr = *my_array;
e) ptr = *my_array[0];

Pointer and Arrays

```
int my_array[] = {1, 2, 3, 4, 5, 6};
int *ptr;
```

Which of the following assignment is correct?

a) ptr = &my_array[0];
b) ptr = my_array;
c) ptr = my_array[0];
d) ptr = *my_array;
e) ptr = *my_array[0];

When you pass array references around, the program keeps track of the array via the address of the first element and the type stored in the array.

This way, when you want the nth element in the array, the system knows to start at the address and jump n*sizeof(type) bytes to get the nth element.

(See next slide for more details)

Pointer Arithmetic

int arr[25];
arr[i] == *(arr + i)

1000				1004				1008			
arr[0] (arr)				arr[1] (arr + 1)				arr[2] (arr + 2)			

We assume that each int takes up 4 bytes of memory. If arr is at address 1000, then arr + 1 is address 1004, arr + 2 is address 1008, and in general, arr + i is address 1000 + (i * 4).
Why is arr + 1 at address 1004 and not 1001? That's pointer arithmetic in action. arr + 1 points to one element past arr. arr + 3 is points to 3 elements past arr. Thus, arr + i points to i elements past arr. The idea is to have arr + i point to i elements after arr regardless of what type of element the array holds.

What is the output?

```
#include<stdio.h>
int main()
{
    int arr[] = {10, 20, 30, 40, 50, 60};
    int *ptr1 = arr;
    int *ptr2 = arr + 5;
    printf("%d", (ptr2 - ptr1));
    printf("%d",  (*ptr2 - *ptr1);
    return 0;
}
```

A. 4, 40
B. 4, 50
C. 5, 5
D. 5, 40
E. 5, 50

What is the output?

```
#include<stdio.h>
int main()
{
    int arr[] = {10, 20, 30, 40, 50, 60};
    int *ptr1 = arr;
    int *ptr2 = arr + 5;
    printf("%d", (ptr2 - ptr1));
    printf("%d",  (*ptr2 - *ptr1);
    return 0;
}
```

A. 4, 40
B. 4, 50
C. 5, 5
D. 5, 40
E. 5, 50

Dereferencing problems

In general, you can make a pointer point anywhere. For example, arr + 1000 is valid, even if the array has only 16 elements. arr - 1000 is also valid. That is, you can compute it, and it won't core dump (crash)!!

However, dereferencing pointers to invalid memory causes problems. Thus, *(arr - 1000) core dumps because you are trying to access the address.

Here's an analogy. You can write down anyone's address on a piece of paper, however, you can't just go inside the person's house at that address (which is like dereferencing). Thus, computing addresses is fine, dereferencing it may cause problems if the address is not a valid address in memory.

Pointer to Pointer

```
int ** dptr;
int *ptr1, *ptr2, *ptr3;
int i = 5, j = 10, k = 20;
ptr1 = &i;
ptr2 = &j;
ptr3 = &k;
```

How to assign values to dptr?

```
dptr =&ptr1;
*dptr = ptr2;
```

Pointer to Pointer

```
int ** dptr;
int *ptr1, *ptr2, *ptr3;
int i = 5, j = 10, k = 20;
ptr1 = &i;
ptr2 = &j;
ptr3 = &k;
```

How to assign values to dptr?

```
dptr =&ptr1;
*dptr = ptr2;
```

Pointers and Memory Allocation (Important for HW9)

```
int * arr;  //pointer to an int
arr = (int*) malloc( 5 * sizeof(int));
```

```
int **arr; //pointer to a pointer
arr = (int **)malloc(5* sizeof(int *)); // arr now is an array of 5 int* pointers
arr[0] = (int*)malloc(5 * sizeof(int));  // arr[0] is now an array of 5 integers.
```

```
void foo( int* p) {
        int j = 5;
        p = &j;
        *p = *p + 4;
        printf("%d", j);
        printf(“%d”, *p);
}
int main(int argc, char * argv[]) {

        int x = 98;
        int *p = &x;
        foo(&x);
        printf(“, %d”, *p);
        printf(", %d", x);
}
```

14. What is the output of the code on the left?

a) 5, 9, 9, 98
b) 5, 9, 98, 9
c) 5, 102, 102, 98
d) 9, 9, 98, 98
e) 9, 9, 9, 98

```
void foo( int* p) {
        int j = 5;
        p = &j;
        *p = *p + 4;
        printf("%d", j);
        printf(“%d”, *p);
}
int main(int argc, char * argv[]) {

        int x = 98;
        int *p = &x;
        foo(&x);
        printf(“, %d”, *p);
        printf(", %d", x);
}
```

14. What is the output of the code on the left?

a) 5, 9, 9, 98
b) 5, 9, 98, 9
c) 5, 102, 102, 98
d) 9, 9, 98, 98
e) 9, 9, 9, 98

What is the output of the following?

```
int main()
{
    char str[20] = "Hello";
    char *p=str;
    *p='M';
    printf("%s\n", str);
    return 0;
}
```

a) Hello
b) Mello
c) M
d) HMello
e) MHello

What is the output of the following?

```
int main()
{
    char str[20] = "Hello";
    char *p=str;
    *p='M';
    printf("%s\n", str);
    return 0;
}
```

a) Hello
b) Mello
c) M
d) HMello
e) MHello

Answer: b)

Infix to Postfix Conversion

Postfix expressions eliminate the need for parentheses. There are no precedence rules to learn, and parenthese are never needed. Because of this simplicity, some compilers and calculators use postfix notation to avoid the complications of multiple sets of parentheses. The operator is placed directly after the two operands it needs to apply.

Examples:

Infix	Postfix
A+B	AB+
A + B * C	A B C * +
(A + B) * C	A B + C *
(A + B) * (C + D)	A B + C D + *

The Conversion Algorithm

- Create an empty stack called operatorStack and an empty string for the output.
- Scan each token (character) in the input string from left to right.
- If the current token is an operand, append it to the end of the output string.
- If the current token is a left parentheses, push it on the stack.
- If the current token is a right parenthesis, keep popping from the stack until the corresponding left parentheses is returned, and append each popped operator to the end of the output string.
- If the current token is an operator, say op1, then,
  while there is an operator token op2 at the top of the operator stack whose precedence is greater than or equal to op1:
  Pop op2 from the stack and append to the end of the output
  Push op1 onto the stack.
- When the input string has been completely processed, pop all the remaining operators from the stack and append them to the end of the output string.
- Return the output string.

Example: A + B * C

current symbol	operator stack	postfix string
A		A
+	+	A
B	+	A B
*	+ *	A B
C	+ *	A B C
		A B C * +

Example: A * (B + C)

current symbol	operator stack	postfix string
A		A
*	*	A
(	* (	A
B	* (	A B
+	* (+	A B
C	* (+	A B C
)	*	A B C +
		A B C + *

Postfix Evaluation

Given an expression AB+C*, how will you evaluate it?