# Plan For The Rest Of The Term

<table>
<thead>
<tr>
<th>Sun</th>
<th>Mon</th>
<th>Tues</th>
<th>Weds</th>
<th>Thurs</th>
<th>Fri</th>
<th>Sat</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3A assigned</td>
<td></td>
<td>3B, 3C assigned</td>
<td>Visit your test, 1015-1230</td>
<td></td>
</tr>
<tr>
<td>Hank OH</td>
<td></td>
<td>3A due</td>
<td>3B due</td>
<td></td>
<td></td>
<td>3C “due”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3D assigned</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12/1</td>
<td>12/2</td>
<td>12/3</td>
<td>12/4</td>
<td>12/5</td>
<td>12/6</td>
<td>12/7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3C extension to this day</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12/8</td>
<td>12/9</td>
<td>12/10</td>
<td>12/11</td>
<td>12/12</td>
<td>12/13</td>
<td>12/14</td>
</tr>
<tr>
<td></td>
<td>Final, 8am-10am</td>
<td></td>
<td>Last day to turn in work, 3D due</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Hank OH**: Holiday
- **3C “due”**: Extension to this day
- **Final, 8am-10am**: Final Exam
Plan For The Rest Of The Term

• 3C: assigned last Thursday -- ~100 lines of code beyond the starter code

• 3D: assigned today
  – Will be major project
  – Will be grade options for partial credit

• No projects after 3D
Sunday OH

- About 7 people attended on Sunday
  - Half the attendance from last year
- Good session, open to doing it again
Hi Everyone,

I have read all of the feedback from Quiz 6 ("how do you feel about the proposed midterm curve?")

There were many responses that were positive. Some talked about the weight on Q4 vs Q5. Others said that I shouldn't be curving at all.

That said, there were some very conscientious responses that are hard to ignore. One said that I did a quiz on strilen (thus increasing the points for that question) ... how much should one question be worth? Others commented that they did poorly on the midterm and that the curve didn't help them.

Overall, I feel that, while the conversation trended towards positive, there was not consensus around my proposed curve.

So, the proposed curve is paused for now, and the conversation will continue.

I do understand there is an issue (with grades being low), and I do want to fix it. But I am not willing to do a curve where everyone gets more points ... in other words, it is very important to me that the grade reflects my true feedback to you. "Everyone gets more points" doesn't provide the targeted feedback I am seeking.

Anyways, I wanted to get this information out there now, so we can have a (short) discussion tomorrow.

Best,
Hank
YouTube Video Ideas?

• (no one sent me email reminders last time)
Implement strdup
Tricky Memory Question
Linked Lists: Motivation

• Will introduce two data structures: linked lists and doubly linked lists
• Linked lists will be used in 3D – you will implement a “map” using linked lists
• Doubly linked lists will tie into a future data structure (Deques)
Linked List (from Sventek)

• A linked list is linear collection of data elements, in which the linear order is not given by their physical placement in memory

• Instead, each node in the list points at the next element in the list.

• Such a structure enables insertion or removal of elements without reallocation or reorganization of the entire structure, since the data items do not need to be stored contiguously in memory
Linked List Example/Analogy
Linked List: possible methods

- InsertBefore
- InsertAfter
- InsertAtBeginning
- InsertAtEnd
- Remove
- Search
Example (from Sventek)

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

struct node *head;

int search(char *university, struct node *theList) {
    struct node *p;

    for (p = theList; p != NULL; p = p->next)
        if (strcmp(p->value, university) == 0)
            return 1;
    return 0;
}
```
How Do Linked Lists Compare With Arrays?
Comparison With Arrays: Hank’s (Wikipedia’s) Answers

- Can do insertions -- beginning, middle, end
- Indexing much slower
- Extra storage

<table>
<thead>
<tr>
<th></th>
<th>Linked list</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Insert/delete at beginning</td>
<td>$\Theta(1)$</td>
<td>N/A</td>
</tr>
</tbody>
</table>
| Insert/delete at end     | $\Theta(1)$ when last element is known;
                            | $\Theta(n)$ when last element is unknown         | N/A   |
| Insert/delete in middle  | search time + $\Theta(1)$[6][7]                  | N/A   |
| Wasted space (average)   | $\Theta(n)$                                      | 0     |
Let’s NOT Implement Linked Lists

• Did this last year, YouTube video:
• https://www.youtube.com/watch?v=B56n1JzvUA0&feature=youtu.be
Doubly Linked List

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

Linked List

```
typedef struct node {
    struct node *next;
    struct node *prev;
    void *value;
} Node;
```

Doubly Linked List

All images from Sventek textbook
(Singly) Linked List Vs Doubly Linked List

I am asked such question and I have my own sayings but I am not really sure what to say about cons and pros? Microsoft asked this question to one of its candidates.

Singly linked list allows you to go one way direction. Whereas doubly linked list has two way direction next and previous.

It all comes down to usage. There's a trade off here.

Singly linked list is simpler in terms of implementation, and typically has a smaller memory requirement as it only needs to keep the forward member referencing in place.

Doubly linked list has more efficient iteration, especially if you need to ever iterate in reverse (which is horribly inefficient with a single linked list), and more efficient deletion of specific nodes.
Example of Doubly Linked List

- iPod play list
  - Need reverse to get to previous song
HashTable

• Idea:
  – Create a big array with keys and values
  – But: don’t insert starting from the beginning
  – Instead: insert into “random” places in the array
  – Not truly random, as it needs to be reproducible
  – Typical: take key and perform some math operation on it

This will be a whirlwind intro to hash tables. We will return to this idea later (NOW).
## HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>
# HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Store:  
UO_ID: 951001234  
Student: “xFF”

Idea:  
Turn UO_ID into an index.  
In this case, %10.  
(much more complex ideas)
### HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>-1</td>
<td>..66</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>NUL</td>
<td>xAF</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Store:
UO_ID: 951003266
Student: “xAF”
# HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>34</td>
<td>44</td>
<td>66</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Store:
- UO_ID: 951012344
- Student: “xAA”

Idea:
- Slot 4 is full ... just use the next slot (slot 5).
- Keep going until you find one, including wraparounds.
## HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>...66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Store:
- UO_ID: 951012345
- Student: “xB8”
### HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>34</td>
<td>44</td>
<td>66</td>
<td>45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Fetch:
UO_ID: 951045323
EASY: NULL
**HashTable**

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key</strong></td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>...66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td><strong>Student</strong></td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Fetch:
UO_ID: 951012345
More work ... walk from index 5 to 6 to 7 ... found it!
# HashTable

## Table

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>...66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

**Fetch:**

UO_ID: 951012355
More work ... walk from index 5 to 6 to 7 to 8... not there!
Another Data Structure for Store/Fetch: Hash Table

typedef struct
{
    int keys[MAX_STUDENTS];
    Student *values[MAX_STUDENTS];
} HashTable;

void Initialize(HashTable *arr)
{
    for (int i = 0 ; i < MAX_STUDENTS ; i++)
    {
        arr->keys[i] = -1;
        arr->values[i] = NULL;
    }
}

void HashTableStore(void *a, int key, Student *v)
{
    HashTable *ht = (HashTable *) a;
    int iter = 0;
    for (int i = 0 ; i < MAX_STUDENTS ; i++)
    {
        int idx = (i+key)%MAX_STUDENTS;
        if (ht->keys[idx] == -1)
        {
            ht->keys[idx] = key;
            ht->values[idx] = v;
            return
        }
    }
    exit(EXIT_FAILURE);
}

Student *HashTableFetch(void *a, int key)
{
    HashTable *ht = (HashTable *) a;
    int iter = 0;
    for (int i = 0 ; i < MAX_STUDENTS ; i++)
    {
        int idx = (i+key)%MAX_STUDENTS;
        if (ht->keys[idx] == key)
        {
            return ht->values[idx];
        }
        if (ht->keys[idx] == -1)
        {
            return NULL;
        }
    }
    return NULL;
}

• Observation:
• Not very generic (int key, Student value)
Complexity for Hash Table

• Store: it depends
  – Things go well: $O(1)$
  – Things go poorly: $O(n)$

• Fetch: it depends
  – Things go well: $O(1)$
  – Things go poorly: $O(N)$

• Gets into new topic ... expected performance.
New Slides: Hash Tables
Hash Tables: Dealing With Collisions

• Collision:
  – HASH(1004) := 4
  – HASH(1014) := 4
  – ... both get mapped to index 4, called “collision”

• How to deal with collision?
  – Open addressing: What we did just now
  – Another idea: “chaining”
  – And more ideas beyond that

Will explore open addressing next, then discuss chaining
Hash Table Taxonomy

Open Addressing:
- Linear Probing
- Quadratic Probing
- Double Hashing
- Other Techniques

Chaining:
- Linked Lists
- Cuckoo Hashing
- Other Techniques

More...

+ choice of hash function
Probing

• When there’s a collision, and you have to find an entry to store it

• “Probing”: how to look for the next open entry

• “Linear probing”: look at the next entry, then the next one, etc.
  – (Our example code)

```c
void HashTableStore(void *a, int key, Student *v)
{
    HashTable *ht = (HashTable *) a;
    int iter = 0;
    for (int i = 0 ; i < MAX_STUDENTS ; i++)
    {
        int idx = (i+key)%MAX_STUDENTS;
        if (ht->keys[idx] == -1)
        {
            ht->keys[idx] = key;
            ht->values[idx] = v;
        }
    }

    exit(EXIT_FAILURE);
}

Student *HashTableFetch(void *a, int key)
{
    HashTable *ht = (HashTable *) a;
    int iter = 0;
    for (int i = 0 ; i < MAX_STUDENTS ; i++)
    {
        int idx = (i+key)%MAX_STUDENTS;
        if (ht->keys[idx] == key)
            return ht->values[idx];
        if (ht->keys[idx] == -1)
            return NULL;
    }

    return NULL;
}
Clustering

• Clustering happens with bad hash functions

• This makes store and fetch slow

• Ideas:
  – Probe more intelligently (next slides)
  – Better hash function (in 10 slides or so)
Quadratic Probing

- Approach for avoiding clusters in hash table
- $H := \text{HASH(key)}$
- Linear probing looks at
  - $H, H+1, H+2, H+3, H+4, \text{ etc.}$
- Quadratic probing looks at
  - $H, H+1, H+4, H+9, H+16, \text{ etc.}$
Double Hashing

- Idea: two hash functions! ... H_1 and H_2
- For key K, apply H_1 to get index H_1(K)
- If H_1(K) is open, then use that index.
- If H_1(K) is not open (collision), then:
  - Apply H_2 to get index H_2(K)
  - Do probing with
    - H_1(K) + H_2(K), H_1(K) + 2* H_2(K), H_1(K) + 3*H_2(K), etc.
  - need to be careful you cover all entries in the table
Hash Table Taxonomy

Open Addressing
- Linear Probing
- Double Hashing
- Other Techniques

Chaining
- Linked Lists
- Other Techniques

Cuckoo Hashing

+ choice of hash function
Chaining: Linked Lists

• (called “separate chaining” on wikipedia)
• Idea: every element in the array is the head of a linked list
• When there is a collision, you put the new item at the front of the linked list
• Some linked lists may get long
• Requires linked list stuff (malloc, extra pointers, etc)
Deletions

• Chaining
  – Easy! Go to the linked list and find the item to remove

• Open Addressing
  – “Lazy Deletion”: don’t delete it, but indicate that it is removed

```c
typedef struct
{
    int keys[MAX_STUDENTS];
    Student *values[MAX_STUDENTS];
    int isValid[MAX_STUDENTS];
} HashTable;
```
The simplest approach to a hash function for character strings is to simply sum the values of the ASCII characters in the string.

```c
long hash1(char *key, long N) {
    unsigned long sum = 0L;
    char *p;

    for (p = key; *p != '\0'; p++)
        sum += (unsigned long)(*p);
    return (long)(sum % N);
}
```
Why Previous Hash Function Was Bad

• Most words are about the same length
• ASCII characters have similar values (‘A’ == 65, ‘Z’ == 90)
• Almost all words will get hash values between 65 and 900
• If we hash a dictionary of 125000 words into a hash table of size 250000, we will get a loss of clustering
Reducing Clustering  
(Polynomial Hash Function)

```c
#define A 31L  
long hash2(char *key, long N) {
    unsigned long sum = 0L;
    char *p;

    for (p = key; *p != '\0'; p++)
        sum = A * sum + (unsigned long)(*p);
    return (long)(sum % N);
}
```

From Sventek Book
## Statistics

<table>
<thead>
<tr>
<th>Function</th>
<th>Total</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Mean</th>
<th>Median</th>
<th>Std Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash1</td>
<td>123,237</td>
<td>0</td>
<td>639</td>
<td>78.8</td>
<td>18</td>
<td>128.3</td>
</tr>
<tr>
<td>hash2(A=1)</td>
<td>123,237</td>
<td>0</td>
<td>639</td>
<td>78.8</td>
<td>18</td>
<td>128.3</td>
</tr>
<tr>
<td>hash2(A=7)</td>
<td>6,072</td>
<td>0</td>
<td>6</td>
<td>0.0522</td>
<td>0</td>
<td>0.261</td>
</tr>
<tr>
<td>hash2(A=19)</td>
<td>806</td>
<td>0</td>
<td>2</td>
<td>0.00650</td>
<td>0</td>
<td>0.0812</td>
</tr>
<tr>
<td>hash2(A=29)</td>
<td>642</td>
<td>0</td>
<td>1</td>
<td>0.00517</td>
<td>0</td>
<td>0.0717</td>
</tr>
<tr>
<td>hash2(A=31)</td>
<td>642</td>
<td>0</td>
<td>1</td>
<td>0.00517</td>
<td>0</td>
<td>0.0717</td>
</tr>
<tr>
<td>hash2(A=37)</td>
<td>642</td>
<td>0</td>
<td>1</td>
<td>0.00517</td>
<td>0</td>
<td>0.0717</td>
</tr>
<tr>
<td>hash2(A=73)</td>
<td>642</td>
<td>0</td>
<td>1</td>
<td>0.00517</td>
<td>0</td>
<td>0.0717</td>
</tr>
<tr>
<td>hash2(A=129)</td>
<td>642</td>
<td>0</td>
<td>1</td>
<td>0.00517</td>
<td>0</td>
<td>0.0717</td>
</tr>
<tr>
<td>hash2(A=257)</td>
<td>642</td>
<td>0</td>
<td>1</td>
<td>0.00517</td>
<td>0</td>
<td>0.0717</td>
</tr>
<tr>
<td>hash2(A=511)</td>
<td>642</td>
<td>0</td>
<td>1</td>
<td>0.00517</td>
<td>0</td>
<td>0.0717</td>
</tr>
</tbody>
</table>
Maps
Maps

• A map stores tuples: (key, value)
• You store both key and value
• You retrieve with just the key (and it gives the value)
• If the key is not in the map, then some appropriate value is returned
Python Equivalent: Dictionaries

```python
>>> dict = {
...    "951301000": "Hank Childs",
...    "951302000": "Zayd Hamoudeh",
...    "951303000": "Priya Kudva",
...    "951304000": "Viet Lai"
...  }

>>> dict["951301000"]
'Hank Childs'
>>> dict["951301001"]
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: '951301001'
```
Functions for the Map Data Structure

• Store
  – Textbook: put

• Fetch
  – Textbook: get

• Other noteworthy functions from textbook:
  – containsKey
  – putUnique
  – isEmpty
  – remove
How To Implement a Map

• Idea #1: Linked Lists
• Idea #2: Hash Tables
How To Implement a Map: Linked Lists

• Make a tuple type (key, value)
• Make a linked list of that tuple type
• Store: insert the tuple into the list
• Fetch: iterate over the linked list, comparing the desired key with the key element of each tuple in the linked list

```c
struct ID_Name_Tuple
{
    int    UO_id;    /* NOTE: Python example used string, not int */
    char   *student_name;
};
```
How To Implement a Map: Hash Tables

• Make a hash function to convert keys to indices
• Decide on hash table details (open addressing, chaining)
• Store: hash key, insert tuple into hash table
• Fetch: hash desired key, compare with key in tuple (repeat as appropriate)
3D

• Implement two types of maps
  – Linked list
  – Hash table

• Analyze performance and write very short report (one paragraph)
## Operator Precedence

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>++ --</td>
<td>Suffix/postfix increment and decrement</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td>( )</td>
<td>Function call</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[ ]</td>
<td>Array subscripting</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>Structure and union member access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-&gt;</td>
<td>Structure and union member access through pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(type){list}</td>
<td>Compound literal(C99)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>++ --</td>
<td>Prefix increment and decrement</td>
<td>Right-to-left</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
<td>Unary plus and minus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>! ~</td>
<td>Logical NOT and bitwise NOT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(type)</td>
<td>Type cast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>Indirection (dereference)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>Address-of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sizeof</td>
<td>Size-of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>__alignof</td>
<td>Alignment requirement(C11)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>* / %</td>
<td>Multiplication, division, and remainder</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>4</td>
<td>+ -</td>
<td>Addition and subtraction</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&lt;&lt;= &gt;&gt;&gt;</td>
<td>Bitwise left shift and right shift</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;= &gt;=</td>
<td>For relational operators &lt; and ≤ respectively</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For relational operators &gt; and ≥ respectively</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>== !=</td>
<td>For relational = and ≠ respectively</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>&amp;</td>
<td>Bitwise AND</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>^</td>
<td>Bitwise XOR (exclusive or)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>Bitwise OR (inclusive or)</td>
</tr>
<tr>
<td>11</td>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>=</td>
<td>Simple assignment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+= -=</td>
<td>Assignment by sum and difference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*= /= %=</td>
<td>Assignment by product, quotient, and remainder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;= &gt;&gt;=</td>
<td>Assignment by bitwise left shift and right shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;= ^ =</td>
<td>Assignment by bitwise AND, XOR, OR</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>,</td>
<td>Comma</td>
<td>Left-to-right</td>
</tr>
</tbody>
</table>

DRAM vs NV-RAM

- **DRAM**: Dynamic Random Access Memory
  - stores data
  - each bit in separate capacitor within integrated circuit
  - loses charge over time and must be refreshed
  - → volatile memory

- **NV-RAM**: Non-Volatile Random Access Memory
  - stores data
  - information unaffected by power cycle
  - examples: Read-Only Memory (ROM), flash, hard drive, floppy drive, ...
Seagate Expansion 5TB Desktop External Hard Drive USB 3.0 (STEB5000100)

by Seagate

$133.99

Get it by Friday, Nov 20

More Buying Choices

$133.99 new (68 offers)

$117.24 used (1 offer)

Crucial Ballistix Sport 16GB Kit (8GBx2) DDR3 1600 MT/s (PC3-12800) UDIMM Memory BLS2KIT8G3D1609DS1S00/ BLS2CP8G3D1609DS1S00

by Crucial

$74.99

Get it by Thursday, Nov 19

More Buying Choices

$69.95 new (73 offers)

Crucial 16GB Kit (8GBx2) DDR3/DDR3L-1600 MHz (PC3-12800) CL11 204-Pin SODIMM Memory for Mac CT2K8G3S160BM / CT2C8G3S160BM

by Crucial

$72.99

Get it by Thursday, Nov 19

More Buying Choices

$71.29 new (99 offers)

$62.00 used (8 offers)

Corsair Vengeance 16GB (2x8GB) DDR3 1600 MHz (PC3 12800) Desktop Memory (CMZ16GX3M2A1600C10)

by Corsair

$83.90

Get it by Thursday, Nov 19

More Buying Choices

$72.50 new (101 offers)

$74.99 used (3 offers)
Relationship to File Systems

• File Systems could be implemented in DRAM.
• However, almost exclusively on NV-RAM
  – Most often hard drives
• Therefore, properties and benefits of file systems are often associated with properties and benefits of NV-RAM.
## DRAM vs NV-RAM properties

<table>
<thead>
<tr>
<th>Property</th>
<th>DRAM</th>
<th>NV-RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>~10GB</td>
<td>~10TB</td>
</tr>
<tr>
<td>Cost</td>
<td>$5/GB</td>
<td>$0.03/GB</td>
</tr>
<tr>
<td>Latency</td>
<td>&lt;100 nanoseconds</td>
<td>~10 milliseconds</td>
</tr>
</tbody>
</table>

What does 100000:1 mean?

Distance: a 20” map of Oregon is 1:100,000 scale

Time: 1 second to 27 hours is 1:100,000 scale

Time: 1 minute to 69 days is 1:100,000 scale

Time: 1 hour to 11 years is 1:100,000 scale

Time: 1 day to 273 years is 1:100,000 scale