Lecture 17: Hash Tables, Maps, Finish Linked Lists
Plan For The Rest Of The Term (1/2)

• 3C: due today
• 3D: assigned tomorrow
  – Major project, worth 8%
  – Grade options for partial credit
• No projects after 3D
Plan For The Rest Of The Term (2/2)

- Lecture 18: November 27, more data structures & sorting
- Lecture 19: November 29, finals review & Hank’s unsolicited advice
- Final: Monday Dec 3rd, 8am, in this room
Oversleeping for the final

- From class syllabus:
  
  “If you miss the Final exam period, you may take the Final afterwards for half-credit. Exceptions for full credit will be granted in appropriate situations. These situations include medical emergencies, etc, and do not include oversleeping, forgetting the day of the final, etc.”
Revising syllabus

• If you run out of late passes, then you may continue to earn half credit on any project up until we take the Final. Once we get to the Final (Monday December 3rd, 2018 at 8am), I will no longer accept homeworks.

• →

• If you run out of late passes, then you may continue to earn half credit on any project up until we take the Final. Once we get to Wednesday December 5th (meaning: Thursday December 6th, 6am) I will no longer accept homeworks.

  – Note: if you misread this statement and think you have until Thursday, there will be no sympathy (don’t ask).
Lab This Week

- No Lab This Week!
OH this week

• Weds?
• Sun?
YouTube Video Ideas?

• (CHECK) 3C overview
• (CHECK) gdb
• More?
Repeat Slide: Hash Tables
From OH

if (strncmp(&rv[0], "R", sizeof(char)))
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>eFF</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>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: 951012345
More work ... walk from index 5 to 6 to 7 ... found it!
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: 951012355
More work ... walk from index 5 to 6 to 7 to 8... not there!
Another Data Structure for Store/Fetch: Hash Table

```c
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
- Other Techniques

More...

Cuckoo Hashing

+ 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_ERROR);
}

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
  – \( H, H+1, H+2, H+3, H+4, \ldots \)
• Quadratic probing looks at
  – \( H, H+1, H+4, H+9, H+16, \ldots \)
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

More...

+ 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 lost 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

• Assigned tomorrow, due Friday November 30th
  – Note: full credit up to Weds December 5th
  – 0% credit as of 6am Thurs Dec 6th
  – (As before, no sympathy for misreading this)
3D

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

• Analyze performance and write very short report (one paragraph)
More on Linked Lists
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 a 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
Example (from Sventek)

```c
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>
<tr>
<td>Insert/delete at end</td>
<td>$\Theta(1)$ when last element is known; $\Theta(n)$ when last element is unknown</td>
<td>N/A</td>
</tr>
<tr>
<td>Insert/delete in middle</td>
<td>search time + $\Theta(1)$[6][7]</td>
<td>N/A</td>
</tr>
<tr>
<td>Wasted space (average)</td>
<td>$\Theta(n)$</td>
<td>0</td>
</tr>
</tbody>
</table>
Let’s Implement Linked Lists
Doubly Linked List

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

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

Linked List

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
Let’s Implement Doubly Linked Lists
Misc. Topics
Unions

• Union: special data type
  – store many different memory types in one memory location

```c
typedef union
{
    float x;
    int y;
    char z[4];
} cis330_union;
```

When dealing with this union, you can treat it as a float, as an int, or as 4 characters.

This union has 4 bytes
Unions

Why are unions useful?

```c
#include <stdio.h>

typedef union
{
    float x;
    int y;
    char z[4];
} cis330_union;

int main()
{
    cis330_union u;
    u.x = 3.5; /* u.x is 3.5, u.y and u.z are not meaningful */
    u.y = 3; /* u.y is 3, now u.x and u.z are not meaningful */
    printf("As u.x = %f, as u.y = %d\n", u.x, u.y);
}
```

```
As u.x = 0.000000, as u.y = 3
```
typedef struct
{
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE PLATE;

typedef struct
{
    char letters[3];
    int nums[3];
} OR_LICENSE PLATE;

typedef struct
{
    int nums[6];
} WY_LICENSE PLATE;

typedef union
{
    CA_LICENSE PLATE ca;
    OR_LICENSE PLATE or;
    WY_LICENSE PLATE wy;
} LicensePlate;
typedef struct
{
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
    char letters[3];
    int nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
    int nums[6];
} WY_LICENSE_PLATE;

typedef union
{
    CA_LICENSE_PLATE ca;
    OR_LICENSE_PLATE or;
    WY_LICENSE_PLATE wy;
} LicensePlate;

typedef enum
{
    CA,
    OR,
    WY
} US_State;

typedef struct
{
    char *carMake;
    char *carModel;
    US_State state;
    LicensePlate lp;
} CarInfo;

int main()
{
    CarInfo c;
    c.carMake = "Chevrolet";
    c.carModel = "Camaro";
    c.state = OR;
    c.lp.or.letters[0] = 'X';
    c.lp.or.letters[1] = 'S';
    c.lp.or.letters[2] = 'Z';
    c.lp.or.nums[0] = 0;
    c.lp.or.nums[1] = 7;
    c.lp.or.nums[2] = 5;
}
# 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;</td>
<td>Bitwise left shift and right shift</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;= &gt;=</td>
<td>For relational operators ≤ and ≥ respectively</td>
<td></td>
</tr>
<tr>
<td></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>13[NOTE 1]</td>
<td>?:</td>
<td>Ternary conditional[NOTE 2]</td>
<td>Right-to-Left</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>=</td>
<td>Assignment by bitwise AND, XOR, and OR</td>
</tr>
<tr>
<td>15</td>
<td>,</td>
<td>Comma</td>
<td>Left-to-right</td>
</tr>
</tbody>
</table>