Sravya Dandibhatta
CS408 Fall 2024
Project 1 - Due Sept 10th

Question 1

Findings: In the comparison of modulo operations across different programming languages, I observed varying results for -5 % 2.

C/C++: The result is -1. Both languages follow the truncated division rule, where the result follows the sign of the dividend.

Java: The result is -1. Java, similar to C/C++, uses truncated division for modulo operations.

Perl: The result is 1. Perl adheres to mathematical conventions where the result is always non-negative. But Perl returns -1 when running on the mc17 architecture.

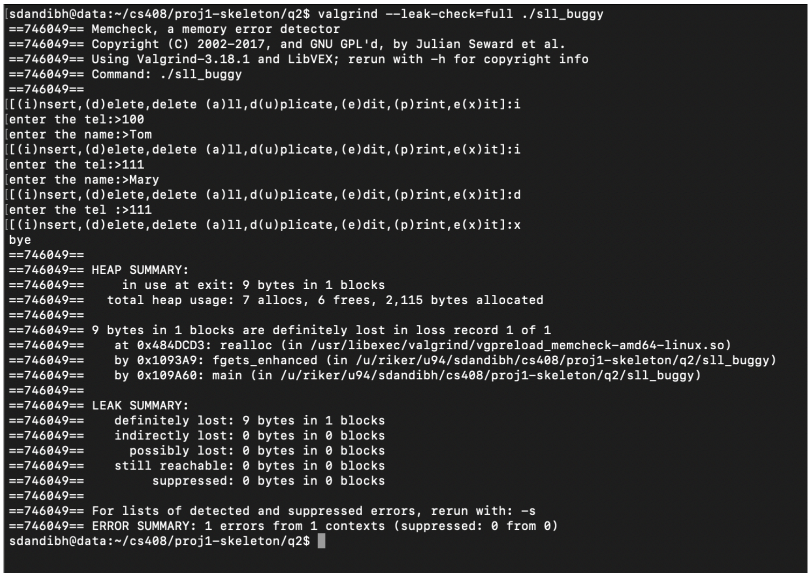
Python: The result is 1. Python also follows mathematical conventions for modulo operations, ensuring the result is non-negative.

Strategies to Cope with Modulo Operation Discrepancies:

1. We can push for a universal rule that all programming languages can follow for modulo operations. This means we must make sure that every language handles the result in the same way, so there’s no confusion.
2. Making sure developers know how different languages handle modulo operations. Having good documentation and tutorials can help them understand these differences and write code that works correctly in any language.
3. Adding features to programming tools that alert developers when their code might have any issues due to how different languages handle modulo. For example, a compiler could warn you if your modulo operation might give different results in different languages.

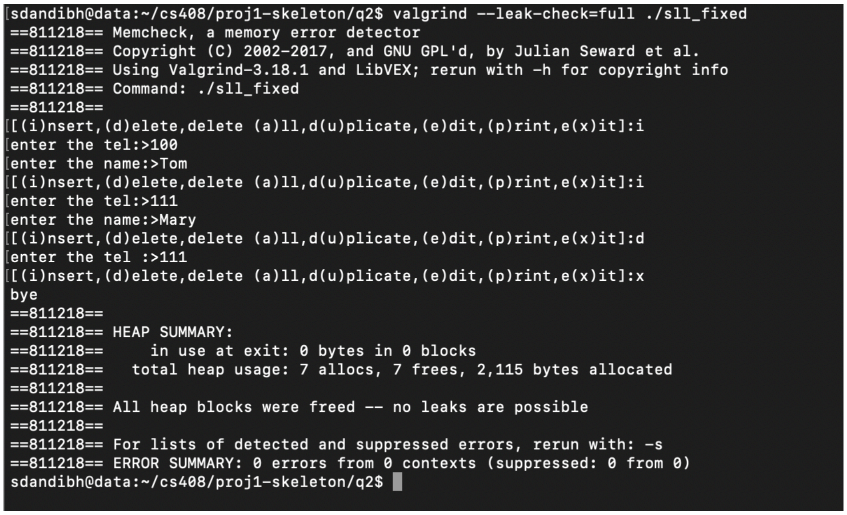
Question 2a

When running valgrind on test case 1:



```
[sdandibh@data:~/cs408/proj1-skeleton/q2$ valgrind --leak-check=full ./sll_buggy
==746049== Memcheck, a memory error detector
==746049== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==746049== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==746049== Command: ./sll_buggy
==746049==
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
[enter the tel:>100
[enter the name:>Tom
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
[enter the tel:>111
[enter the name:>Mary
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:d
[enter the tel :>111
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
bye
==746049==
==746049== HEAP SUMMARY:
==746049==     in use at exit: 9 bytes in 1 blocks
==746049==   total heap usage: 7 allocs, 6 frees, 2,115 bytes allocated
==746049==
==746049== 9 bytes in 1 blocks are definitely lost in loss record 1 of 1
==746049==    at 0x484DCD3: realloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==746049==    by 0x1093A9: fgets_enhanced (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==746049==    by 0x109A60: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==746049==
==746049== LEAK SUMMARY:
==746049==    definitely lost: 9 bytes in 1 blocks
==746049==    indirectly lost: 0 bytes in 0 blocks
==746049==      possibly lost: 0 bytes in 0 blocks
==746049==    still reachable: 0 bytes in 0 blocks
==746049==         suppressed: 0 bytes in 0 blocks
==746049==
==746049== For lists of detected and suppressed errors, rerun with: -s
==746049== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
sdandibh@data:~/cs408/proj1-skeleton/q2$
```

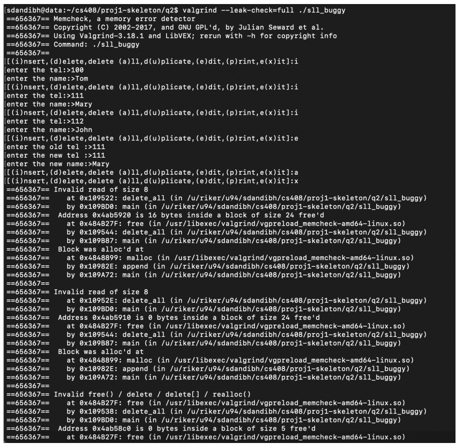
The Valgrind output indicates a memory leak of 9 bytes in the fgets_enhanced function. This leak occurs because memory allocated with realloc is not properly freed before the program exits, leading to a block of memory that remains allocated but is no longer accessible. To resolve this issue, ensure that any dynamically allocated memory is properly freed before the program terminates. Specifically, check all paths in the code where memory is allocated and ensure that corresponding free calls are made to release this memory before exiting. We make the necessary fix in the delete_node() function where we make sure to free(temp->str). And when we run Valgrind on the new sll_fixed.c file with updated code, we get that there are no leaks.



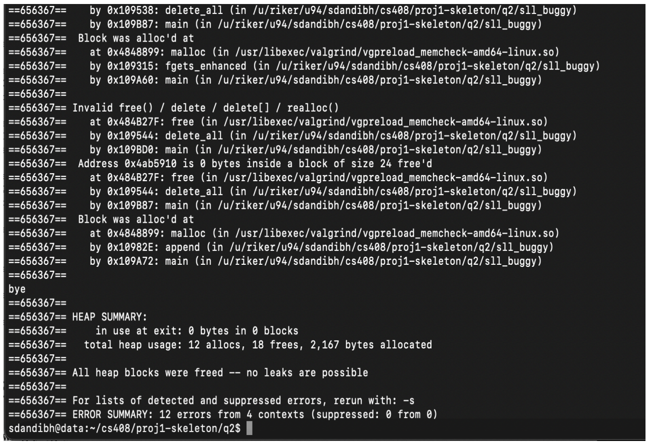
```
[sdandibh@data:~/cs408/proj1-skeleton/q2$ valgrind --leak-check=full ./sll_fixed
==811218== Memcheck, a memory error detector
==811218== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==811218== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==811218== Command: ./sll_fixed
==811218==
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
[enter the tel:>100
[enter the name:>Tom
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
[enter the tel:>111
[enter the name:>Mary
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:d
[enter the tel :>111
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
bye
==811218==
==811218== HEAP SUMMARY:
==811218==     in use at exit: 0 bytes in 0 blocks
==811218==   total heap usage: 7 allocs, 7 frees, 2,115 bytes allocated
==811218==
==811218== All heap blocks were freed -- no leaks are possible
==811218==
==811218== For lists of detected and suppressed errors, rerun with: -s
==811218== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
sdandibh@data:~/cs408/proj1-skeleton/q2$
```

Question 2b

When running valgrind on test case 2:

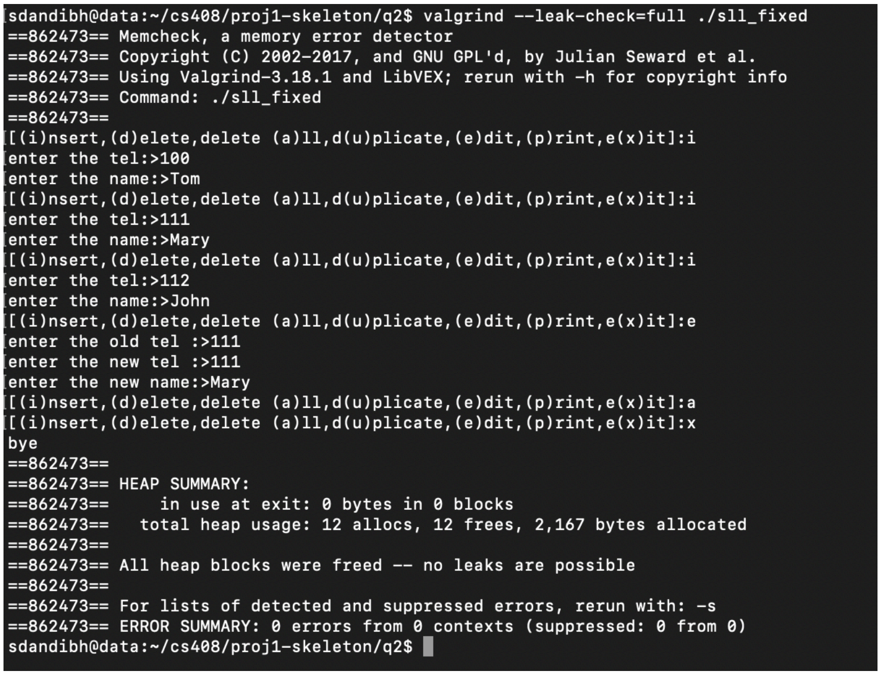


```
[sdandibh@data:~/cs408/proj1-skeleton/q2$ valgrind --leak-check=full ./sll_buggy
==656367== Memcheck, a memory error detector
==656367== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==656367== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==656367== Command: ./sll_buggy
==656367==
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
[enter the tel:>100
[enter the name:>Tom
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
[enter the tel:>111
[enter the name:>Mary
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
[enter the tel:>112
[enter the name:>John
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:e
[enter the old tel :>111
[enter the new tel :>111
[enter the new name:>Mary
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:a
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
==656367== Invalid read of size 8
==656367==    at 0x109522: delete_all (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109BD0: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==  Address 0x4ab5920 is 16 bytes inside a block of size 24 free'd
==656367==    at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==656367==    by 0x109544: delete_all (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109B87: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==  Block was alloc'd at
==656367==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==656367==    by 0x10982E: append (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109A72: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==
==656367== Invalid read of size 8
==656367==    at 0x10952E: delete_all (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109BD0: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==  Address 0x4ab5910 is 0 bytes inside a block of size 24 free'd
==656367==    at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==656367==    by 0x109544: delete_all (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109B87: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==  Block was alloc'd at
==656367==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==656367==    by 0x10982E: append (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109A72: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==
==656367== Invalid free() / delete / delete[] / realloc()
==656367==    at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==656367==    by 0x109538: delete_all (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109BD0: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==  Address 0x4ab58c0 is 0 bytes inside a block of size 5 free'd
==656367==    at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
```



```
==656367==    by 0x109538: delete_all (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109B87: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==  Block was alloc'd at
==656367==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==656367==    by 0x109315: fgets_enhanced (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109A60: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==
==656367== Invalid free() / delete / delete[] / realloc()
==656367==    at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==656367==    by 0x109544: delete_all (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109BD0: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==  Address 0x4ab5910 is 0 bytes inside a block of size 24 free'd
==656367==    at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==656367==    by 0x109544: delete_all (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109B87: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==  Block was alloc'd at
==656367==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==656367==    by 0x10982E: append (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==    by 0x109A72: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==656367==
bye
==656367==
==656367== HEAP SUMMARY:
==656367==     in use at exit: 0 bytes in 0 blocks
==656367==   total heap usage: 12 allocs, 18 frees, 2,167 bytes allocated
==656367==
==656367== All heap blocks were freed -- no leaks are possible
==656367==
==656367== For lists of detected and suppressed errors, rerun with: -s
==656367== ERROR SUMMARY: 12 errors from 4 contexts (suppressed: 0 from 0)
sdandibh@data:~/cs408/proj1-skeleton/q2$
```

The issue: The Valgrind output reveals two primary issues in the delete_all function. First, there's an invalid read of size 8, where the function attempts to access memory that has already been freed, specifically trying to access the next attribute of a node that has been deallocated. To fix this, ensure that pointers to freed memory are set to NULL immediately after freeing the memory. Second, there's an invalid free or double-free error, indicating that the function is trying to free memory that has already been freed. This often happens if the same block of memory is freed more than once. To address this, add a check in the delete_all function to ensure that a pointer is not freed multiple times, by verifying that the pointer is not NULL before calling free. We make the necessary fix in delete_all() function where we set p to NULL. After making the change and running valgrind on sll_fixed.c, we see there are no leaks.



```
[sdandibh@data:~/cs408/proj1-skeleton/q2$ valgrind --leak-check=full ./sll_fixed
==862473== Memcheck, a memory error detector
==862473== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==862473== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==862473== Command: ./sll_fixed
==862473==
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
[enter the tel:>100
[enter the name:>Tom
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
[enter the tel:>111
[enter the name:>Mary
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
[enter the tel:>112
[enter the name:>John
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:e
[enter the old tel :>111
[enter the new tel :>111
[enter the new name:>Mary
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:a
[[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
bye
==862473==
==862473== HEAP SUMMARY:
==862473==     in use at exit: 0 bytes in 0 blocks
==862473==   total heap usage: 12 allocs, 12 frees, 2,167 bytes allocated
==862473==
==862473== All heap blocks were freed -- no leaks are possible
==862473==
==862473== For lists of detected and suppressed errors, rerun with: -s
==862473== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
sdandibh@data:~/cs408/proj1-skeleton/q2$
```

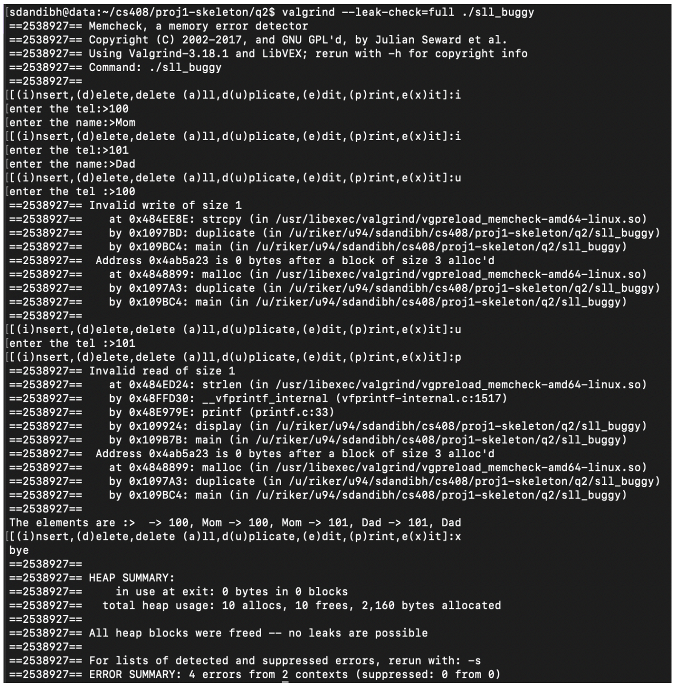
Question 2c

The test case I came up with:

```
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>100
enter the name:>Mom
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>101
enter the name:>Dad
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:u
enter the tel :>100
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:u
enter the tel :>101
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:p
The elements are :>  -> 100, Mom -> 100, Mom -> 101, Dad -> 101, Dad
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
bye
```

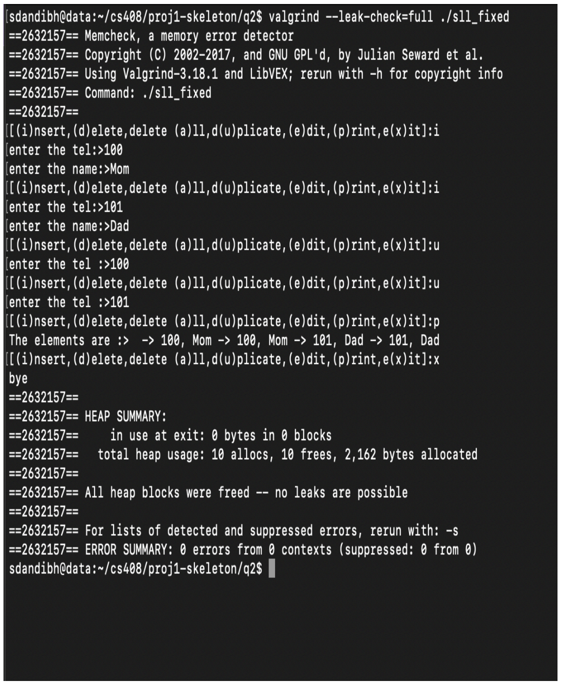
The duplicate function did not allocate enough memory for the string, causing invalid memory access errors. I increased the memory allocation size by adding 1 byte to account for the null terminator in the duplicate function, and I was able to solve the issue. This change ensures that the string has enough space to include the null terminator, preventing memory access errors when duplicating nodes.

When running my test case on sll_buggy.c:



```
sdandibh@data:~/cs408/proj1-skeleton/q2$ valgrind --leak-check=full ./sll_buggy
==2538927== Memcheck, a memory error detector
==2538927== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2538927== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2538927== Command: ./sll_buggy
==2538927==
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>100
enter the name:>Mom
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>101
enter the name:>Dad
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:u
enter the tel :>100
==2538927== Invalid write of size 1
==2538927==    at 0x484EE8E: strcpy (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==2538927==    by 0x1097BD: duplicate (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==2538927==    by 0x109BC4: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==2538927==  Address 0x4ab5a23 is 0 bytes after a block of size 3 alloc'd
==2538927==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==2538927==    by 0x1097A3: duplicate (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==2538927==    by 0x109BC4: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==2538927==
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:u
enter the tel :>101
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:p
==2538927== Invalid read of size 1
==2538927==    at 0x484ED24: strlen (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==2538927==    by 0x48FFD30: __vfprintf_internal (vfprintf-internal.c:1517)
==2538927==    by 0x48E979E: printf (printf.c:33)
==2538927==    by 0x109924: display (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==2538927==    by 0x109B7B: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==2538927==  Address 0x4ab5a23 is 0 bytes after a block of size 3 alloc'd
==2538927==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==2538927==    by 0x1097A3: duplicate (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==2538927==    by 0x109BC4: main (in /u/riker/u94/sdandibh/cs408/proj1-skeleton/q2/sll_buggy)
==2538927==
The elements are :>  -> 100, Mom -> 100, Mom -> 101, Dad -> 101, Dad
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
bye
==2538927==
==2538927== HEAP SUMMARY:
==2538927==     in use at exit: 0 bytes in 0 blocks
==2538927==   total heap usage: 10 allocs, 10 frees, 2,160 bytes allocated
==2538927==
==2538927== All heap blocks were freed -- no leaks are possible
==2538927==
==2538927== For lists of detected and suppressed errors, rerun with: -s
==2538927== ERROR SUMMARY: 4 errors from 2 contexts (suppressed: 0 from 0)
```

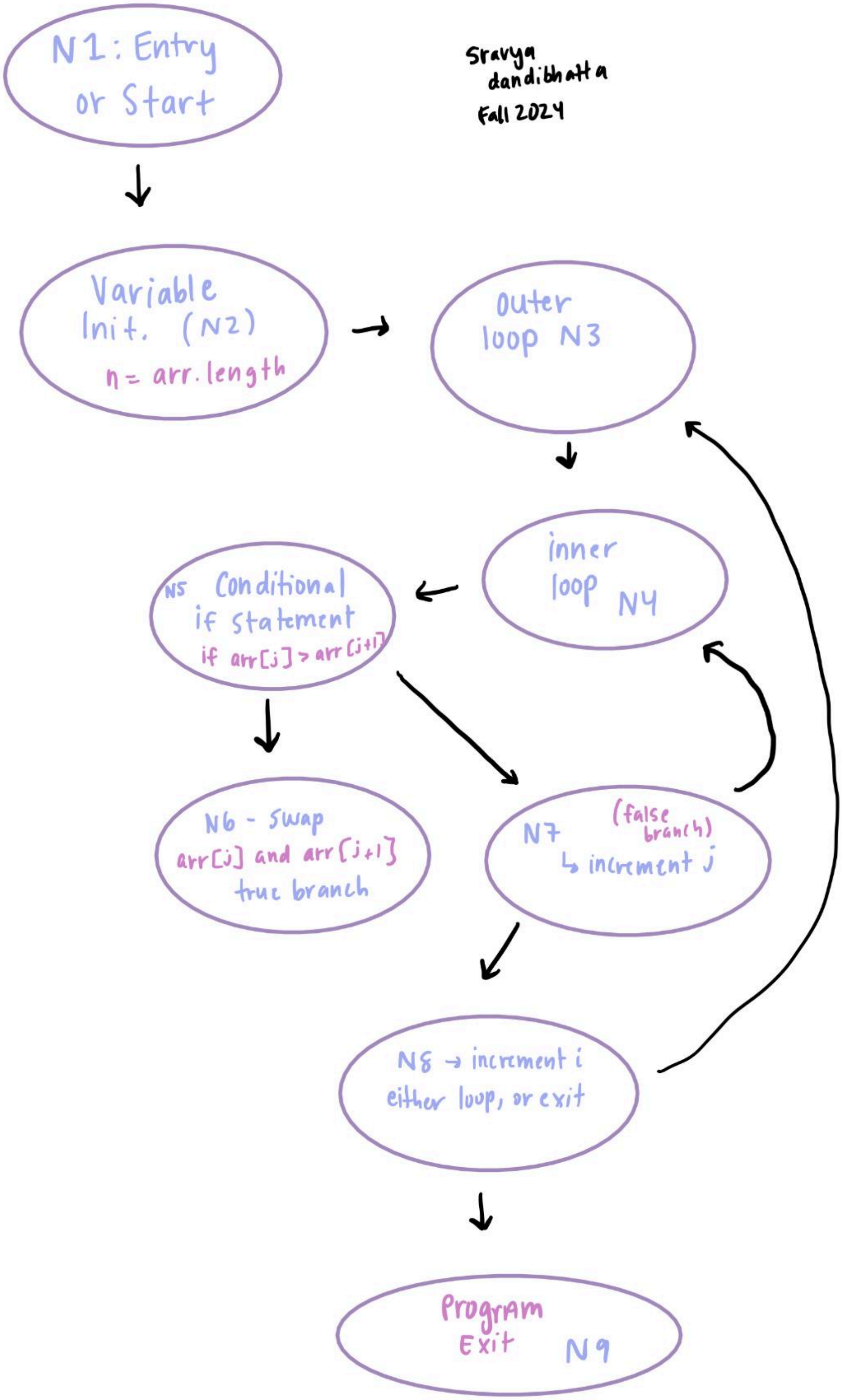
When running my test case on sll_fixed.c:



```
sdandibh@data:~/cs408/proj1-skeleton/q2$ valgrind --leak-check=full ./sll_fixed
==2632157== Memcheck, a memory error detector
==2632157== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2632157== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2632157== Command: ./sll_fixed
==2632157==
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>100
enter the name:>Mom
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>101
enter the name:>Dad
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:u
enter the tel :>100
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:u
enter the tel :>101
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:p
The elements are :>  -> 100, Mom -> 100, Mom -> 101, Dad -> 101, Dad
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
bye
==2632157==
==2632157== HEAP SUMMARY:
==2632157==     in use at exit: 0 bytes in 0 blocks
==2632157==   total heap usage: 10 allocs, 10 frees, 2,162 bytes allocated
==2632157==
==2632157== All heap blocks were freed -- no leaks are possible
==2632157==
==2632157== For lists of detected and suppressed errors, rerun with: -s
==2632157== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
sdandibh@data:~/cs408/proj1-skeleton/q2$
```

Question 3

Sravya
dandibhatta
Fall 2024



Question 4a

Node Coverage (TR_{NC}) - To achieve node coverage, we must ensure that each node in the CFG is executed at least once. The test requirements for node coverage are:

- TR_{NC}: [N1], [N2], [N3], [N4], [N5], [N6], [N7], [N8], [N9], [N10], [N11]

Edge Coverage (TR_{EC}) - To achieve edge coverage, we need to ensure that each edge in the CFG is traversed at least once. The test requirements for edge coverage are:

- TR_{EC}: [N1 → N2], [N1 → N3], [N2 → N3], [N3 → N4], [N3 → N5], [N3 → N6], [N3 → N7], [N4 → N8], [N5 → N8], [N6 → N7], [N7 → N8], [N8 → N9], [N8 → N10], [N9 → N11], [N10 → N11]

Edge-Pair Coverage (TR_{EPC}) - To achieve edge-pair coverage, you must cover all possible pairs of edges. The test requirements for edge-pair coverage are:

- TR_{EPC}: [N1 → N2 → N3], [N1 → N3 → N4], [N1 → N3 → N5], [N1 → N3 → N6], [N1 → N3 → N7], [N2 → N3 → N4], [N2 → N3 → N5], [N2 → N3 → N6], [N2 → N3 → N7], [N3 → N4 → N8], [N3 → N5 → N8], [N3 → N6 → N7], [N3 → N7 → N8], [N4 → N8 → N9], [N4 → N8 → N10], [N5 → N8 → N9], [N5 → N8 → N10], [N6 → N7 → N8], [N7 → N8 → N9], [N7 → N8 → N10], [N8 → N9 → N11], [N8 → N10 → N11]

Prime Path Coverage (TR_{PPC}) - To achieve prime path coverage, you need to cover all prime paths, which are the longest paths that do not repeat nodes. The test requirements for prime path coverage are:

- TR_{PPC} (Reachable): [N1 → N2 → N3 → N6 → N7 → N8 → N9 → N11], [N1 → N2 → N3 → N5 → N8 → N9 → N11], [N1 → N2 → N3 → N4 → N8 → N10 → N11], [N1 → N2 → N3 → N7 → N8 → N9 → N11], [N1 → N3 → N6 → N7 → N8 → N9 → N11], [N1 → N3 → N4 → N8 → N10 → N11]
- TR_{PPC} (Unreachable): [N1 → N2 → N3 → N6 → N7 → N8 → N10 → N11], [N1 → N2 → N3 → N5 → N8 → N10 → N11], [N1 → N2 → N3 → N4 → N8 → N9 → N11], [N1 → N3 → N6 → N7 → N8 → N10 → N11], [N1 → N2 → N3 → N7 → N8 → N10 → N11], [N1 → N3 → N4 → N9 → N11], [N1 → N3 → N5 → N8 → N10 → N11], [N1 → N3 → N7 → N8 → N10 → N11]

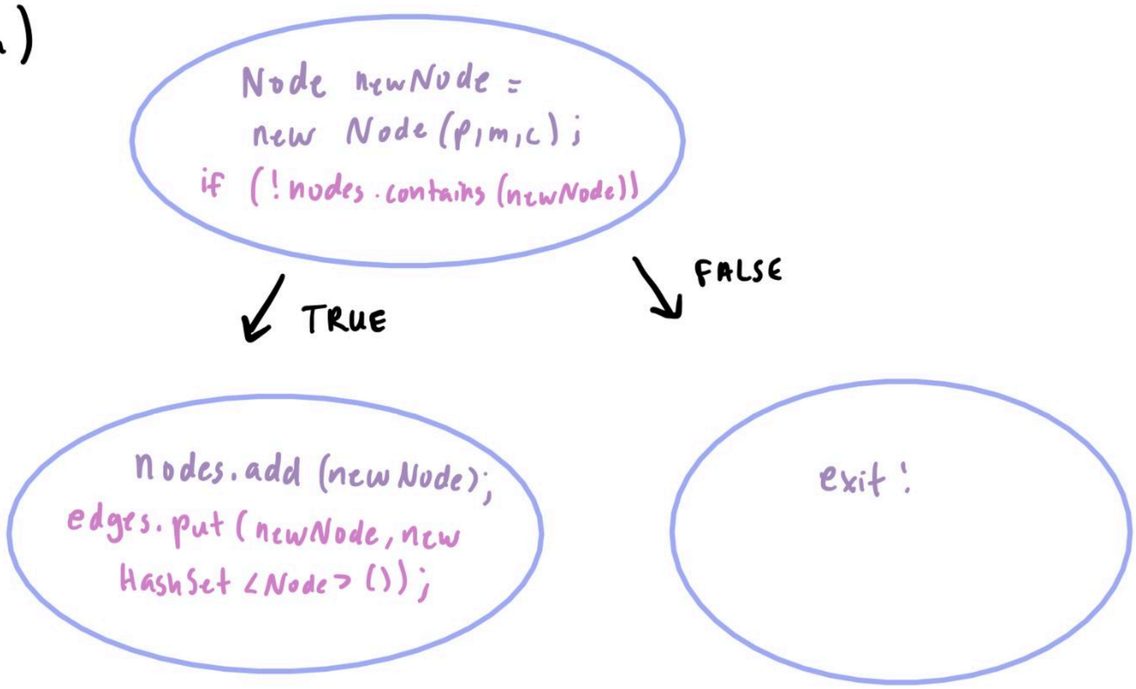
Question 4b

For this, the test cases in TestM-skeleton.java make sure we have node coverage, meaning all the nodes in the control flow graph get visited at least once, but not necessarily all the edges connecting them. Then, we aim for edge coverage, where every edge gets visited at least once, but not all possible edge pairs are hit, that’s where edge-pair coverage comes in. However, in this case, it's impossible to achieve edge-pair coverage without also covering prime paths since any set of tests that fully covers all edge pairs will also end up covering all the prime paths. Lastly, for prime path coverage, the test cases are designed to ensure all the longest, non-repeating paths in the graph are tested. So basically, each type of coverage builds on the previous one, and it’s hard to get some without getting others.

Question 5a

The test class satisfies both node coverage (NC) and edge coverage (EC) for the addNode method. The control flow graph for addNode consists of three nodes: method entry, the check if the node already exists, and the node addition with edge initialization. The addNode() test case covers the path where a new node is added, traversing all three nodes. The addNode_duplicate() test case covers the path where an existing node is not added, hitting the first two nodes. Together, these test cases visit all nodes and traverse all edges in the graph, thus satisfying both NC and EC.

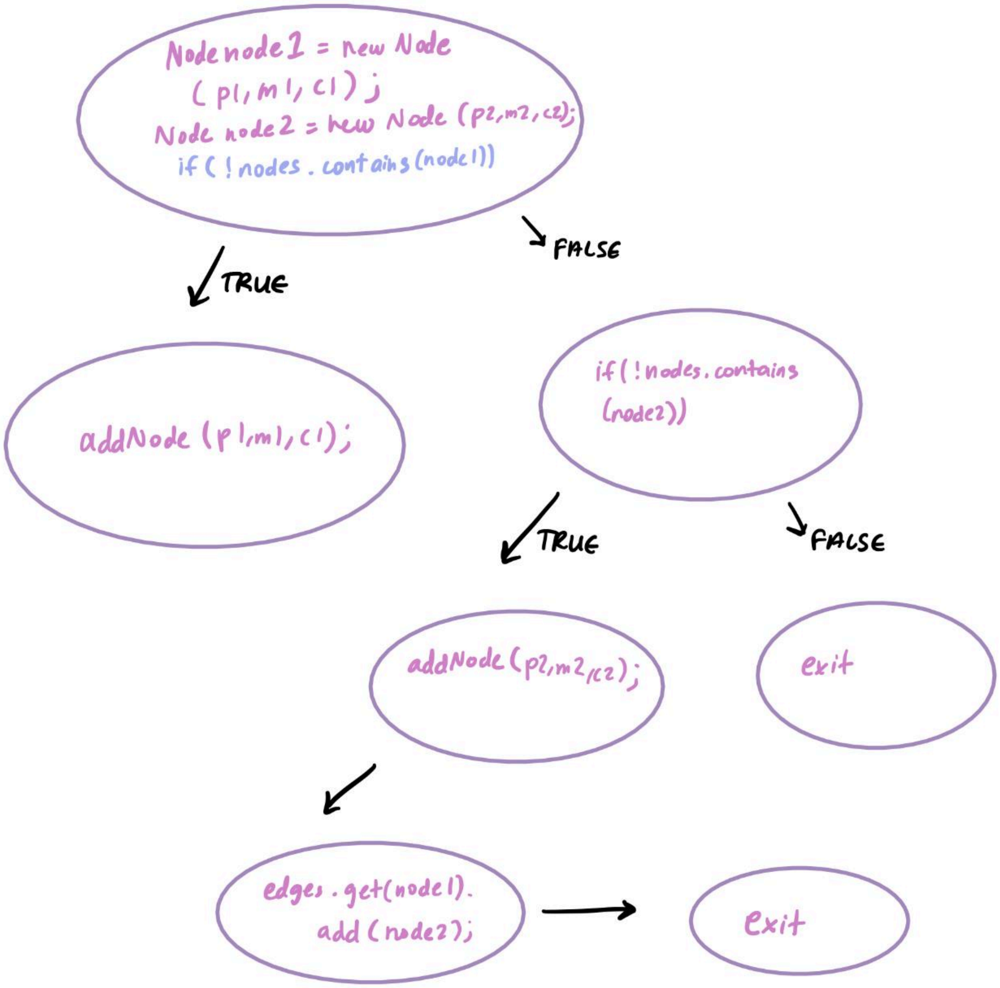
5a)



Question 5b

For the addEdge method, the test class satisfies NC. The control flow graph includes nodes for method entry, checking if each node exists, adding nodes if they don't exist, and adding the edge. The addEdge() test case covers the scenario where both nodes need to be added before creating the edge, traversing all nodes in the graph. The addEdge_existingNodes() test case covers the path where both nodes already exist and only the edge needs to be added. These two test cases combined cover all nodes and all possible edges in the control flow graph, meeting the criteria for both NC.

5b)

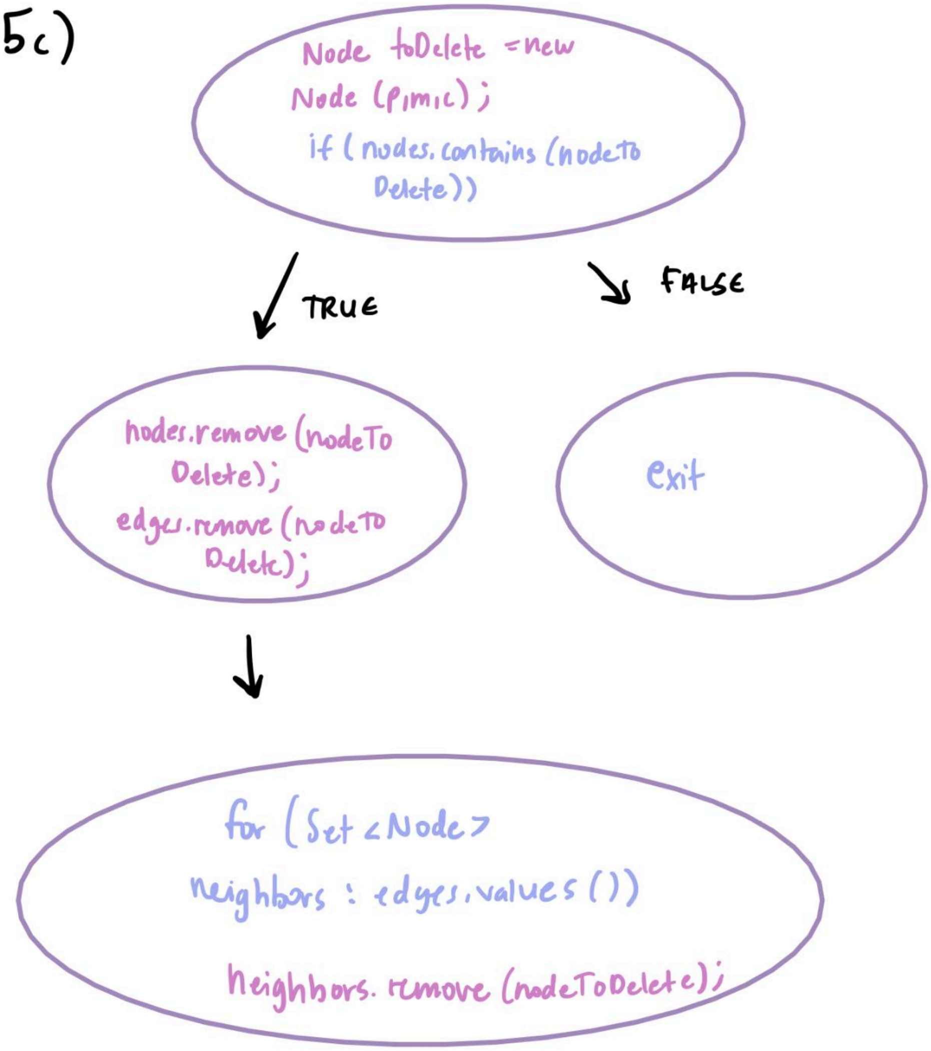


Node node1 = new Node (p1, m1, c1);
Node node2 = new Node (p2, m2, c2);
if (!nodes.contains(node1))
TRUE
FALSE
addNode (p1, m1, c1);
if (!nodes.contains (node2))
TRUE
FALSE
addNode (p2, m2, c2);
exit
edges.get(node1).add (node2);
exit

Question 5c

The test class satisfies both Node Coverage and edge coverage for the deleteNode method. The control flow graph for this method includes nodes for method entry, checking if the node exists, removing the node from the nodes set, removing it from the edges map, and removing it from all neighbor sets. The deleteNode() test case covers the path of deleting an existing node, while deleteNode_nonexistent() covers the case where the node doesn't exist.

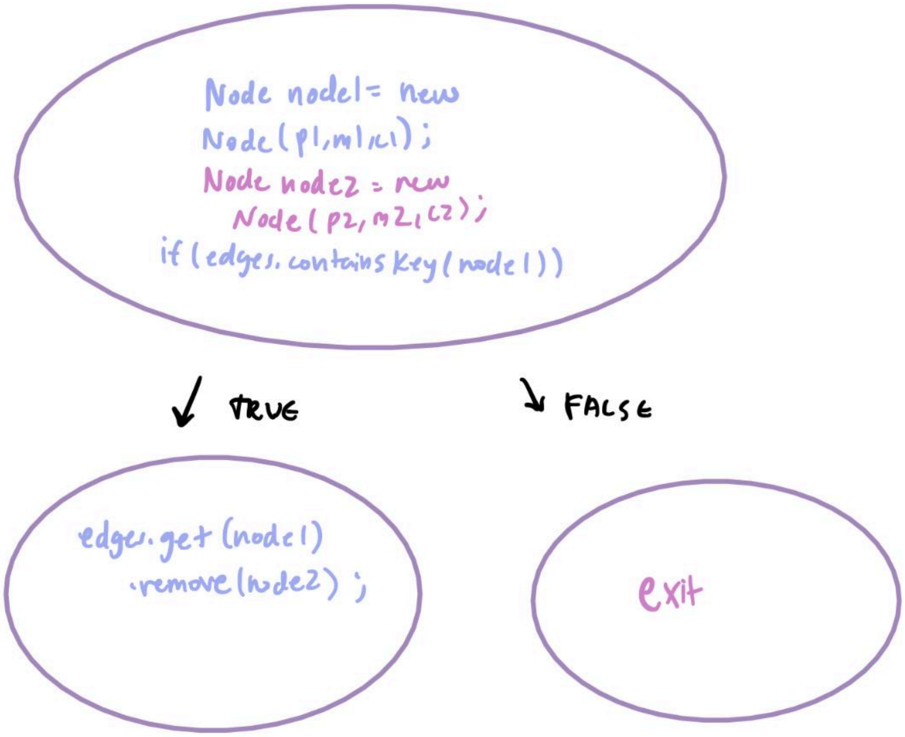
5c)


Node toDelete = new Node (p,m,c);
if (nodes.contains (nodeToDelete))
TRUE
FALSE
nodes.remove (nodeToDelete);
edges.remove (nodeToDelete);
exit
for (Set<Node> neighbors : edges.values())
neighbors.remove (nodeToDelete);

Question 5d

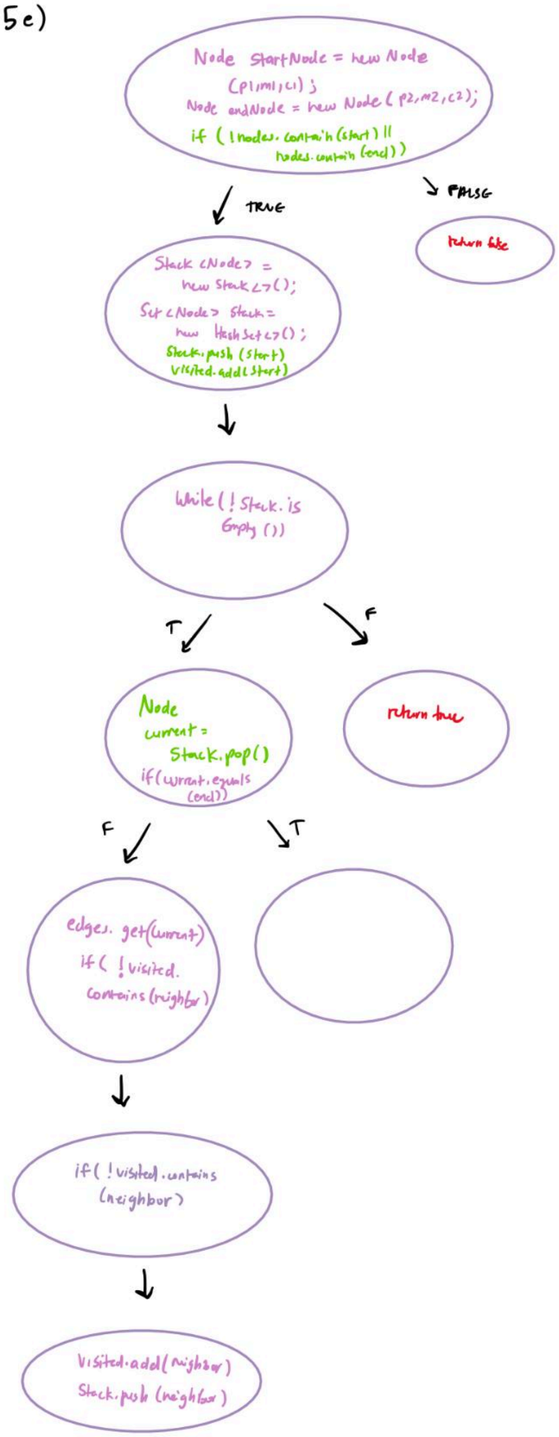
For the deleteEdge method, the test class satisfies both NC and EC. The control flow graph is relatively simple, consisting of nodes for method entry, checking if the first node exists in the edges map, and removing the second node from the first node's neighbors. The deleteEdge() test case covers the scenario where an existing edge is deleted, traversing all nodes in the graph. The deleteEdge_nonexistent() test case covers the path where the edge to be deleted doesn't exist. These two test cases combined cover all nodes and all possible edges in the control flow graph, and they therefore satisy both NC and EC criteria.

5d)



Question 5e

5e)



Question 5f

After analyzing the test coverage for the methods in CFGTest.java, I found that the majority of the methods meet the criteria for Node Coverage and Edge Coverage. The addNode method achieves full NC and EC, since the combination of test cases successfully covers all the possible nodes and edges, including any scenarios where nodes are either newly added, and/or duplicated. The addEdge method, however, only meets the requirements for NC.

The deleteNode method satisfies both NC and EC, because the provided test cases comprehensively cover all nodes and edges, including any cases where a node is deleted or does not exist. The deleteEdge method also meets both NC and EC, as the test cases cover all possible scenarios of edge deletion, whether the edge exists or not. The isReachable method satisfies both NC and EC as well because the test cases ensure that all the necessary nodes and edges within the method's control flow are covered, achieving full coverage.

While most of the methods in CFGTest.java satisfy the requirements for both NC and EC, the addEdge method remains an exception.